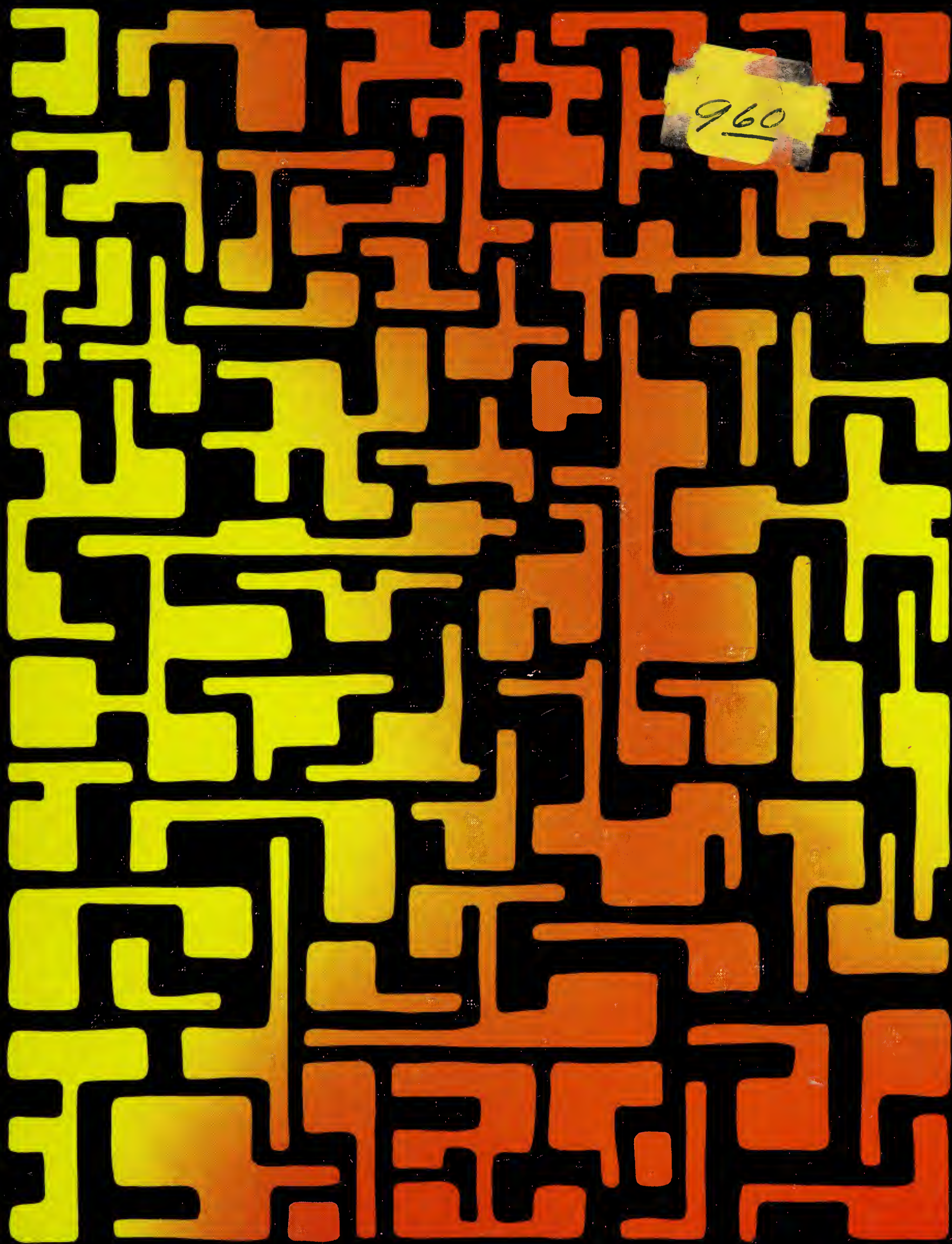


INTRODUCTION TO COMPUTER SCIENCE



PAUL W. MURRILL AND CECILL SMITH



Digitized by the Internet Archive
in 2018 with funding from
Kahle/Austin Foundation

Introduction to Computer Science

Paul W. Murrill

Cecil L. Smith

Louisiana State University

Introduction to Computer Science

Intext Educational Publishers

New York

U-B-RK

Copyright © 1973 by Intext Press, Inc.

All rights reserved. No part of this book may be reprinted, reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Library of Congress Cataloging in Publication Data

Murrill, Paul W

Introduction to computer science.

1. Electronic data processing. 2. Electronic digital computers. I. Smith, Cecil L., joint author.
II. Title.

QA76.5.M849 001.6'4 73-637

ISBN 0-7002-2420-3

INTEXT EDUCATIONAL PUBLISHERS
666 FIFTH AVENUE
NEW YORK, N.Y. 10019

Contents

Preface	vii	2.6	Input/Output	59	
1 Introduction	1	2.7	Hierarchy of Operations in Arithmetic Expressions	60	
1.1	Computers of the Past	2	2.8	Compound Conditionals	63
1.2	Problem Solving and Programming	7	3 Loops and Iteration Counters	79	
1.3	The Language of Flowcharts	8	3.1	Construction of Simple Loops	79
1.4	Programming Languages	10	3.2	Iteration Counters	83
1.5	The Basic Programming Language	13	3.3	Examples of the Use of Loops	89
1.6	The Fortran Programming Language	15	3.4	Nested Loops	93
1.7	Model of a Simple Computing Machine	17	3.5	Debugging	96
1.8	Interpreters and Compilers	20	3.6	Multiple Branching	102
1.9	Debugging	23	4 Subscripted Variables	115	
1.10	Batch-Processing Systems	26	4.1	The Need for Subscripted Variables	115
1.11	Conversational Time-Sharing	31	4.2	Memory Allocation	119
2 Algorithmic Processes	38	4.3	Use of Subscripted Variables	121	
2.1	A Nonnumerical Algorithm	38	4.4	Table Look-Up in an Ordered Array	128
2.2	An Example of a Numerical Algorithm	41	4.5	Input/Output Statements for Arrays	132
2.3	Programming the Algorithm	46	4.6	Use and Misuse of Subscripted Variables	135
2.4	Decision Blocks	52	4.7	Arrays with Multiple Subscripts	139
2.5	Truncation and Rounding	54			

4.8	Processing of Character Data	143
4.9	Linear Lists	150
4.10	One-Way Lists	153

5 Functions and Procedures 174

5.1	Functions	176
5.2	Procedures or Subroutines	183
5.3	Communication with Functions and Procedures	188
5.4	Multiple Entry	194
5.5	Multiple Return	195

6 Computer Hardware 203

6.1	Number Systems	203
6.2	Storage of Information in a Digital Computer	207
6.3	Binary Representation of Numerical Data	210
6.4	Storage of Characters	216
6.5	Decimal Numbers	217
6.6	Machine Instructions	222
6.7	Use of Multiple Accumulators	225
6.8	Indexed Addressing	229
6.9	Assembly-Language Programming	231

7 The Computer System 234

7.1	The Central Processing Unit, Memory, and I/O Support Systems	234
7.2	Processor Features	238
7.3	Peripheral Units	241
7.4	The Use of Compilers	254
7.5	Batch-Processing Systems	257
7.6	Multiprogramming	260
7.7	Time-Sharing	262

8 Trees, Games, and Networks 264

8.1	Concept of a Tree	264
8.2	Representation, Storage, and Search of Trees	268
8.3	A Game	275
8.4	Problems Involving Decisions	280
8.5	A Transportation Problem	282
8.6	The Unwary Traveler	284

9 Compilation 294

9.1	Tree Representation of Expressions	294
9.2	The Polish Notation	296
9.3	An Algorithm for the Translation Process	301
9.4	Arithmetic Assignment Statements	307
9.5	Translating Postfix Expressions to Assembly Code	308
9.6	Finishing the Job	318

10 Business Systems 322

10.1	Organization of Data	323
10.2	Processing of Files	329
10.3	System Charts	340
10.4	External-Sort Algorithms	345

11 Numerical Applications 354

11.1	Finite Arithmetic	354
11.2	Use and Evaluation of Series	358
11.3	Solution of Nonlinear Algebraic Equations	362
11.4	The Golden Section	368
11.5	Integration	375
11.6	Gauss Reduction	382
11.7	Linear Regression	395

Appendix A	Assembly Language Programming	413
-------------------	-------------------------------	-----

Appendix B	Basic	437
-------------------	-------	-----

Appendix C	Fortran	462
-------------------	---------	-----

Appendix D	PL/I	523
-------------------	------	-----

Appendix E	Operations on Binary, Octal, and Hexadecimal Numbers	587
-------------------	--	-----

Appendix F	Conversion Tables	598
-------------------	-------------------	-----

Index	615
-------	-----

Preface

Our primary objective is to introduce college students to computing. The book grew out of our experiences in teaching courses entitled “Introduction to Computer Science,” and it covers the topics required to meet the needs of such a first course.

We have tried to achieve a certain balance among the following: (1) flowcharting and algorithm development (Chapters 2 through 5); (2) the computer system, including hardware and software aspects (Chapters 6 and 7); and (3) applications. In regard to applications, this text provides applications within computer science (such as compiling), applications in the field of business (Chapter 10), and numerical applications (Chapter 11).

The text is designed for those who would like an in-depth introduction to computer science, as opposed to a superficial coverage of the more glamorous topics. As such, the text is designed for course B-1, Introduction to Computing, in the suggested computer science curriculum appearing in “Curriculum 68.”

In order to be language independent we have emphasized flowcharting. Since most students taking this course also will be learning a programming language, appendixes are included on Basic, Fortran, and PL/I. For other languages, the text must be supplemented by a programming manual. We have also chosen to introduce in Appendix A two subsets of the assembly languages used on commercial computers, namely, those on the IBM System/360 and the XDS Sigma 5, along with macros or subroutine calls to handle I/O and other similar operations that are too complex to be taught in an introductory course. These macros and subroutines are available in the solutions manual along with an emulator written in Fortran IV for the IBM subset. This subset has been successfully used for several years now, and it offers the advantage of allowing students with initiative to proceed to other assembly language instructions if they so wish. In effect, the simple instructions can be used without having to learn the complex I/O statements, yet the power of the full assembler is available if the student wants to use it. The authors wish to acknowledge the efforts of Dr. Walter G. Rudd and Dr. Geoffrey D. Creede in developing this approach.

In writing the text the authors have made certain chapters independent of previous chapters to permit the instructor sufficient flexibility in order of presentation. In this respect the following list of chapter “prerequisites” might be useful:

Chapter	Prerequisite chapter
1	—
2	1
3	2
4	3
5	4
6	1
7	6
8	4
9	4 (8 helpful)
10	4 (5 helpful)
11	4 (5 helpful)

In effect, Chapters 6 and 7 can be scheduled at any point. All subsequent chapters depend primarily on Chapter 4.

The authors wish to acknowledge several individuals who have contributed to making this book a reality. We would like to thank Mr. Alan Krigman, editor of *Instruments and Control Systems*, for providing source material for the sections in Chapter 1 treating the development of computers. The authors also wish to thank the secretarial staffs of the Department of Computer Science, Department of Chemical Engineering, and Office of Academic Affairs at Louisiana State University, Baton Rouge, for their assistance. Thanks is also expressed to Misses Cathy Barrett and Nina Stewart for proofreading, but the authors must still assume the responsibility and apologize for those, hopefully few, errors that remain. We are also gratified by Mr. Bill Michaels’ efforts on the solutions manual. Finally, our deepest appreciation goes to the many students who lived with us through the development and refinement of the material presented.

Introduction to Computer Science



INTRODUCTION

Although any evaluation of the significance of technological developments is certainly open to criticism, one factor that must be considered is the impact on the everyday life of the average person. On this point, the development of the digital computer must rank as one of the most significant developments of modern times. At the time of this writing, the computer has influenced today's society far more than many other truly important developments of this century, including atomic energy.

The impact of the computer upon today's society is vividly summarized in the following paragraph taken from the book *Future Shock* by Alvin Toffler[†]:

The computer burst upon the scene around 1950. With its unprecedented power for analysis and dissemination of extremely varied kinds of data in unbelievable quantities and at mind-staggering speeds, it has become a major force behind the latest acceleration in knowledge-acquisition. Combined with other increasingly powerful analytical tools for observing the invisible universe around us, it has raised the rate of knowledge-acquisition to dumbfounding speeds.

The development of the computer has been so rapid that models in use in the early 1960s can be considered as antiques. By now, everyone has seen a card marked "do not fold, spindle, or mutilate."

Although, as we shall see in the next section, the history of the concept of digital computers can be traced back several centuries, they first became of practical utility during the 1940s. At that time, a man with a desk calculator could perform something on the order of 1,000 computations per day. Today, commercially available computers can perform over 1 million operations per second. Thus, our computational capability during this period has increased by a factor of over 20 million or, in other words, over seven orders of magnitude.

Without computational capability of this type, man would never have gone to the moon and returned by 1970. In fact, a special-purpose digital computer actually made the trip with him. The numerical (or mathematical) applications

[†]Random House, Inc., New York, © 1970.

of digital computers in all scientific disciplines have been extensive and may very well have been one of the prominent factors that made possible the significant push to make engineering a science as opposed to an art. Today, large computers are used for hours to obtain the solution to complex numerical problems, a process often referred to as “number-crunching.”

One must not, however, think of the digital computer as only a “fast desk calculator.” It is much more than this. Whereas an individual must continuously monitor and direct the conventional desk calculator to perform each computation, the digital computer can accept a complete set of instructions and then follow them obediently without further human direction. This ability to direct itself through even the most complicated set of logic distinguishes the computer from the desk calculator.

Whereas the desk calculator is limited to numerical operations, the digital computer is not. Translation of natural languages, for example, Russian to English, is coming within the capability of today’s machines. Computers have been expert checker players for some time and are getting quite good at chess. They can analyze electrocardiograms and can recognize handwritten characters (except mine). It is quite possible that these nonnumerical applications may one day become the most important service provided by the digital computer.

Before proceeding too far in a text of this type, perhaps we should define a computer. Stated simply, a *computer* is a system that mechanizes the processing of information. Unfortunately, this definition is considerably broader than is reflected by the common use of the term *computer*. Under this definition, desk calculators, the abacus, the slide rule, and similar devices would classify as computers. In practice, the term *computer* is generally meant to designate a “stored-program digital computer.” One objective of the remainder of this chapter is to define what is meant by “stored-program” and “digital.”

1.1. COMPUTERS OF THE PAST

As we shall see, computers, or “systems that mechanize the processing of information,” date back many years. Generally, these systems can be classified in one of the following categories:

1. Logic machines
2. Digital devices
3. Analog devices

In this section, we shall attempt to give only a few examples of computers and not a complete development in chronological order of computers as we know them today. We shall first examine an early logic machine and then look at digital and analog devices.

Being a computer in the broadest sense of the term, the *Ars Magna* was a seminal form of logic machine, supposedly fabricated after a divine illumination by the thirteenth-century Spanish theologian, Ramon Lull. The device consisted

of concentric geometric figures, on whose peripheries were inscribed basic principles or categories in various branches of knowledge. Rotation of the figures created combinations of ideas for contemplation. For example, the nine-segment circle inscribed with the attributes of God (goodness, eternity, etc.), could be rotated relative to the circle of basic questions (why, how great, etc.). This yielded such provocative questions as “Why is God good?” or “How great is his eternity?” In the most complex of Lull’s machines, the multicolored *figura universalis*, fourteen concentric wheels provided topics of a staggering number and complexity.

Lull believed that his methods would be instrumental in converting the heathens to Christianity and was devoted to this missionary work. Although he was stoned to death in North Africa by an Arab mob, his devices became the basis of an enduring philosophy. Known as the Doctor Illuminatus, Lull was supported by the Franciscans but regarded as insane by the Dominicans. He was influential enough after three centuries to be worthy of derision by Francis Bacon as the creator of “a mass and heap of the terms of all arts . . . that has ends of everything but nothing of worth.”

Another device that would be classified by our definition as a computer is the *abacus*, a wooden frame strung with parallel wires on which beads were moved to indicate amounts. The abacus is a digital device in that it operates on numbers generally expressed directly in the decimal number system, although the abacus principle could be used with other number systems. Prior to the abacus arithmetic operations were accomplished using pebbles (Latin, *calculi*) placed in grooves, again constituting a digital method.

In the tenth century, the monk Gerbert (later Pope Sylvester II) became familiar with Arabic and Hindu number systems. He noted the correspondence of these systems with the abacus and accordingly designed a counter on which the beaded wires were replaced by wheels circumscribed with numerals. Gerbert avoided the Saracen figures but adopted the Hindu numerals because he reasoned that these people were not necessarily infidels. It was eventually realized that the wheel was not necessary and that calculations could be made directly with the written decimal numbers. By the thirteenth century, the abacus had therefore been relegated in Europe to the illiterate classes.

Mechanical aids to computation reappeared in 1617 with the *numbering rods* of John Napier. The rectangular faces of flat wooden sticks were divided lengthwise into nine sections inscribed with multiples of various digits by factors from 1 to 9. The rods contained the multiplication tables and also served the memory function necessary for carrying figures in products of multidigit numbers.

Logarithms, also invented by Napier, were the basis for the *line of numbers* devised by Edmund Gunter in 1620. This was a scale subdivided in proportion to the logarithms of integers between 1 and 10. Addition or subtraction of lengths, accomplished with a pair of compasses, yielded segments equivalent to the product or quotient of the components. Gunter also devised scales proportional to the logarithms of trigonometric functions for use in navigation. William

Oughtred combined two Gunter scales into a slide rule to simplify manipulation. Circular and spiral forms had appeared by 1650.

On the slide rule, lengths are used to represent values. To multiply two numbers, lengths (proportional to the logarithms of the two numbers) are added to give the product. Any length can be accommodated, so long as the length does not exceed that of the slide rule (which cannot occur for multiplication). Thus, the slide rule is an *analog* device as opposed to a *digital* device.

The first authentically documented calculator, built in 1642 by Blaise Pascal, was an automated version of the counter devised by Gerbert seven centuries earlier. Numbers from 0 to 9 were engraved on a series of geared input wheels, positioned to present decimal places. When these were turned, sum or difference digits on output wheels appeared in slots. An elaborate mechanism was provided to carry decimal places. The *machina arithmetica*, conceived by Leibniz in 1671 and completed in 1694, was a similar device capable of automatic multiplication. Leibniz at one time also believed that the *Ars Magna* of Ramon Lull contained the basis of the rational machine in the form of a symbolism and technique from which a complete description of the universe might be formulated.

Tabular methods had evolved during the eighteenth century to simplify evaluation of complex functions. Charles Babbage built a *difference engine* in 1822 which could work to six decimal places and produce such tables automatically. He obtained government support for a twenty-place machine, but the project was subject to countless delays and was eventually abandoned.

In 1833 Babbage conceived the *analytical engine*, a computer which would operate without human intervention at each step. Initial data and operating instructions would be transferred from punched cards to geared wheels in a memory, or store. Values would be exchanged during calculations with computing wheels in a mill through gears and levers. The steam-driven engine was to be capable of one addition per second and would produce a direct printout.

Babbage understood the scientific implications of computation, and envisioned chemistry, for example, as “a branch of mathematical analysis, which . . . would enable us to predict the character of any new compound, and . . . the source from which its formation might be indicated.” The extreme mechanical complexity and the ambitious scope of the project mitigated against its success, however, as did Babbage’s lack of administrative ability, his intolerance, and his embittered personality. A friend remarked that Babbage had not had a single completely happy day in his life and probably hated mankind in general, Englishmen in particular, and the English government and organ grinders most of all. After Babbage’s death the analytical engine fell into obscurity, and for over a century calculating machines were in the tradition of Pascal and Leibniz.

In 1937, Harvard graduate student Howard Aiken proposed a general-purpose digital computer and with assistance from the IBM Corporation completed the *Automatic Sequence Controlled Calculator Mark I* in 1944. The device used mechanical counters driven through relay-controlled electromagnetic clutches and contained over 760,000 wheels and relays as well as 500 miles of wire.

Decimal additions and subtractions could be performed in one-third of a second, multiplications typically required 5 seconds, and divisions averaged 16 seconds.

In 1939 George Stibitz deduced that telephone relays could be employed to multiply or divide complex numbers. His computer, using binary-coded-decimal arithmetic, was demonstrated in 1940 with a remote teletype input/output station. Similar relay machines were employed to calculate fire-control instrument adjustments during World War II, and several general-purpose models were built subsequently.

J. Presper Eckert and John Mauchly began work in 1943 at the University of Pennsylvania on an electronic device to compute firing tables for ballistic trajectories. The resulting Electronic Numerical Integrator And Calculator (ENIAC) was a decimal computer, with one flip-flop for each possible value of each digit and hand-wired instructions for each program. The 30-ton machine required 18,000 vacuum tubes and covered 1,500 square feet but could perform 5,000 additions per second.

John von Neumann proposed to replace the manually wired ENIAC program with a stored program using a coded prewired instruction set. A routine could then be implemented as a series of numerical steps and adaptively modified during operation. Binary arithmetic, advocated by Leibniz and applied to electronic counting by C.E. Wynn-Williams in 1932, was also suggested by von Neumann. These concepts, first applied in 1949 to the EDVAC at the University of Pennsylvania and the EDSAC at Cambridge, became the basis for present digital-computer design.

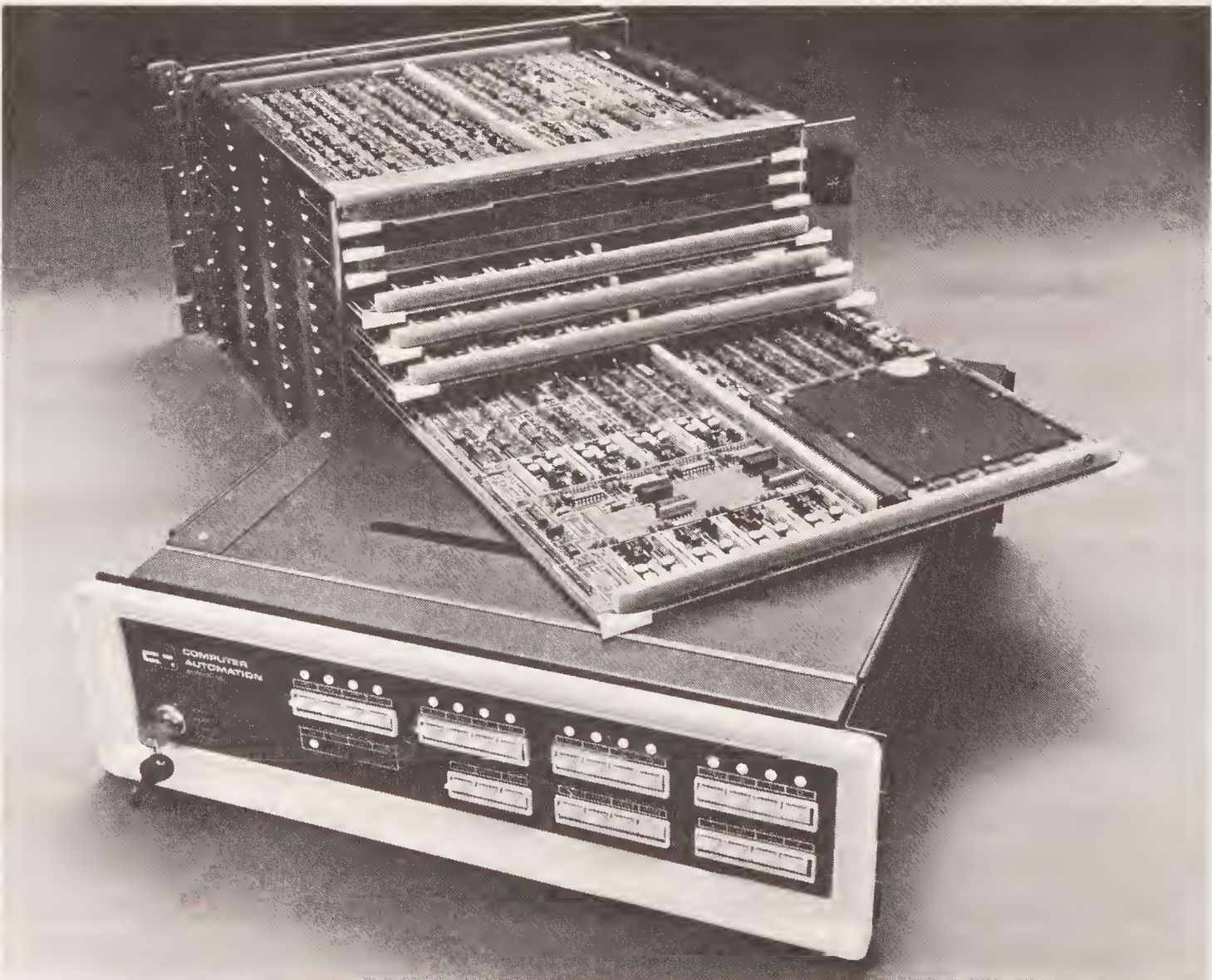
Following von Neumann's concept of the stored program, most of the advances in computing technology can be traced to the incorporation of solid-state technology and other developments in electronics, many of which are byproducts of space technology. Speeds have increased to the point where current commercial computers can perform 1 million additions per second. Today, medium- to large-scale digital computers have storage capacities of a million characters or more implemented in the form of magnetic cores. Information can be written into or read from these storage units in one-millionth of a second or less.

In addition to core storage units, auxiliary storage units in the form of disks or drums are now popular. In these devices, information is stored on magnetically coated surfaces rotating under a read/write head. In disks, the surface is on a rotating flat plate; in drums, the surface is the outside of a cylinder. Although the time required (that is, the access time) to read or write information on these devices is considerably longer than for magnetic cores, the storage capacity is much larger and the cost is substantially less. For example, a medium-scale centralized computing facility for a university will have an auxiliary storage capacity of 50 million characters or more.

Currently, computers are available in a wide range of sizes. At the upper end are machines such as the Burroughs B-6700 (Figure 1.1a), costing several million dollars but supporting the work of hundreds of people. At the other end are



a. Large computer system (Courtesy Burroughs Corporation)



b. Minicomputer

Figure 1-1 Computing Systems

computers like the Alpha 16 (Figure 1.1b) that can be purchased for less than \$10,000. While having less storage capacity and less in the way of input/output devices, there are some tasks that this computer can actually perform as fast as the Burroughs B-6700. The smaller machines are generally purchased for a single purpose, whereas the larger machines provide the versatility to meet a wide range of requirements.

1.2. PROBLEM SOLVING AND PROGRAMMING

In introductory courses in computer science, the student's basic objective is to learn how to use the computer to solve problems. In gaining proficiency at this undertaking, the student must understand the general nature and characteristics of digital computers. It is not necessary, however, that he understand all the processes that proceed in the internals of the computing machine while it is solving his problem. Is it necessary to understand the process of food digestion in order to eat?

Before tackling a specific problem to be solved on a computer, we might first examine the general procedure followed in solving problems with the aid of a digital computer. When a problem for which a solution is to be obtained has been presented, an appropriate course of action could be as follows:

1. Study the problem carefully, making sure that you understand the statement of the problem and what is desired. Obtain any additional information or data that are needed in order to solve the problem. Resolve any uncertainties about the problem statement before proceeding.
2. Is the problem appropriate for solving on a digital computer? If so, continue. If not, pursue other avenues. At this point we are admitting that some problems simply do not require or are not appropriate for the digital computer.
3. Develop a step-by-step computational procedure that, if followed, will lead to a solution of the problem or to the conclusion that there is no solution for the problem as stated. The digital computer is a *sequential machine*; that is, it performs one step at a time. Problems stated as a sequence of specific actions are quite suitable to being solved on a digital computer, provided, of course, that the actions called for in each step can be performed by the computer.
4. Express the procedure developed in the previous step in whatever manner is most appropriate for the problem and for the computer on which it is to be solved. The result of this step is a set of instructions or procedures, commonly called a *program*, that the computer is to follow in solving the problem.
5. Enter the results of Step 4 into the computer and direct it to perform whatever actions are called for by the program. In the process of following these instructions, the desired answers will be produced by the

computer. In fact, part of the instructions will direct the computer to print whatever information is needed to comprise a solution to the problem.

In practice, this procedure is frequently not a once-through procedure as the above might lead one to believe. For example, the answers produced in Step 5 may be obviously incorrect, meaning that an error was made in one of the previous steps. In a later section, we shall present a more realistic picture of this procedure.

As will be the general practice throughout this book, a specific example will be used to illustrate the concept of problem solving. As numerical problems are generally more readily understood, our problem will be numerical in nature but kept simple so that the pertinent points are not obscured by mathematical complexity.

The problem is stated as follows:

From numerical values for a salesman's weekly pay and his sales for the past week, compute his total weekly pay, which is his weekly base pay plus 4 percent of all weekly sales over \$100.

How would you solve this problem? The following steps lead to the correct answer:

1. If total sales is greater than \$100, compute his commission as 0.04 times his sales less \$100. Otherwise, let his commission be zero.
2. Compute his total pay as his base pay plus commission (from Step 1).

A step-by-step procedure such as the above is generally referred to as an *algorithm*. Deriving an algorithm is the basic objective of Step 3 of the approach to problem solving given at the beginning of this section.

Since the next four chapters of this text will consider various algorithms, we shall address the question of the nature of algorithms in more detail. Basically, an algorithm is a computational procedure that terminates in a finite number of steps. The computational procedure should take into account any possible situation that might arise. Failure to do so will often lead to difficulties in a subsequent step of the problem-solving approach presented at the beginning of this section.

In many respects, the most critical step in problem solving is the development of a correct algorithm. Unfortunately, we shall see that there is no step-by-step procedure for deriving an algorithm. That is, there is no algorithm for deriving algorithms. The only approach is to study algorithms in terms of examples, an approach we shall use in the following chapters.

1.3. THE LANGUAGE OF FLOWCHARTS

In many cases, the nature of the algorithm is more evident from a pictorial representation than from the statement representation of the previous section.

The flowchart is a convenient mechanism for pictorially presenting algorithms. When a computer program is the final objective, operations such as input and output can also be incorporated. Figure 1-2 gives the flowchart that incorporates the algorithm presented in the previous section for computing the salesman's pay.

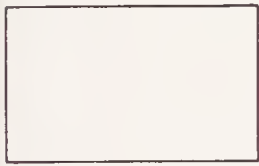
When we draw flowcharts like the one in Figure 1-2, it is helpful to standardize on the shapes for the various boxes. In this text, we shall always use the following conventions:



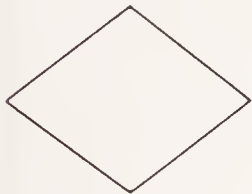
Oval boxes indicate starting or terminal points.



Boxes of this type generally indicate input or output of data in the form of punched cards. We shall use this convention for input of data throughout the text, although many students will probably be using teletype or typewriter terminals.



Rectangular boxes indicate computations or operations on data.



Diamond-shaped boxes indicate decision points.



Boxes of this shape indicate printed output.

These boxes are connected by lines with arrows that indicate the manner of proceeding from one box to another.

Different individuals find flowcharts of varying utility in solving problems. Some draw flowcharts for practically all problems; others rarely draw a flowchart. But regardless of whether or not one chooses to use flowcharts, they are nevertheless an extremely convenient vehicle for portraying concepts in a text such as this one. For this reason, we shall make extensive use of flowcharts, drawing one for practically every problem we examine.

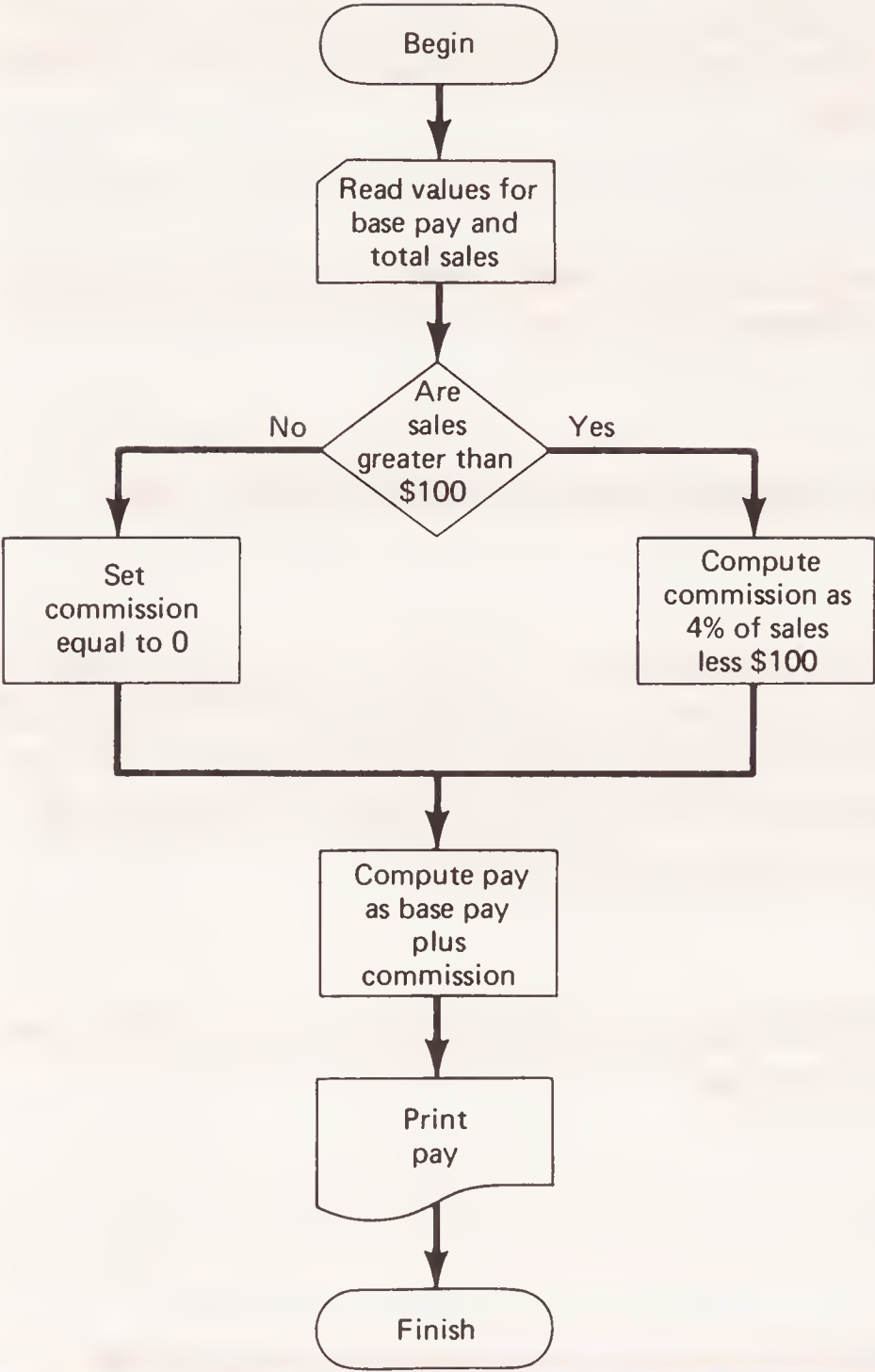


Figure 1-2 Flowchart for computing salesman's pay

1.4. PROGRAMMING LANGUAGES

After an algorithm has been developed and a flowchart prepared, the next step is to develop an explicit set of instructions for the computer to follow in solving the problem. When expressed in a form that is understandable by the computer, this set of instructions is referred to as a *program*. The program must incorporate the algorithm for solving the problem along with the necessary input and output operations that permit entry of the required data and output of the desired answers. The person who prepares such a program is referred to as a *programmer*.

A language is a vehicle for communication. For each language, a set of rules or conventions for construction of sentences must be followed for most effective use of the language. These rules pertaining to sentence formulation, word relationships, and punctuation comprise the grammar, or *syntax*, which differs

from one language to the next. Also, each language has its own alphabet, which consists of the characters acceptable for use in the language.

To date, natural languages such as English have not proven convenient for communicating with computers. Probably the basic problem is that many constructions within natural languages are ambiguous, especially when examined out of context. Furthermore, they are not necessarily the best vehicle for expressing algorithms. For example, to many the flowchart in Figure 1-2 is much more concise and explicit than the verbal statement of the algorithm given in Section 1.2. Natural languages tend to be wordy; algorithms tend to be more direct.

A *programming language* is a language suitable for communicating with a computer. Since natural languages are not currently being used for this purpose, all programming languages in use at present are artificial languages that have been developed specifically for use with computers. To date, a number of such languages have appeared, and the improvement of existing languages and the development of new ones is an active area within computer science.

In this section, we shall restrict our attention to *problem-oriented* programming languages, that is, languages that are designed for use by those who want to solve problems within their respective disciplines. Since in the early chapters of this book we are primarily interested in solving problems, we shall at this point examine languages of this type only. Programming languages designed for other purposes will be discussed in Chapter 6.

Currently a number of different programming languages enjoy widespread use. While there is definite merit to standardizing on a single language for universal use, it must be recognized that different applications have different requirements, which has led to the development of several languages, each with its special characteristics that make it attractive to a class of computer users. For example, scientists and engineers view computers as a tool for solving complex numerical problems and are therefore primarily interested in the ease and flexibility with which algebraic expressions can be expressed in the programming language. Business or commercial users do not normally need such algebraic capabilities but are more interested in the capabilities for processing large quantities of data, for example, printing paychecks from data on each of their employees. For instructional use (especially in beginning courses), the important characteristic is the ease with which enough features can be learned to begin to solve problems. In fact, many disciplines have somewhat unique requirements that could be best met by a programming language specifically designed for that discipline. While a number of different problem-oriented programming languages exist, about a half dozen have enjoyed and will probably continue to enjoy widespread use.

Fortran (Formula Translator). Developed in the mid-1950s by International Business Machines (IBM), this language is by far the most popular language among scientists and engineers and has found considerable applications within the business community. Since its inception, features have been steadily added

to the language. Although more powerful and flexible languages exist, the popularity of Fortran will probably continue for many years.

Cobol (Common Business-Oriented Language). Primarily through the efforts of the Federal government, Cobol was developed in the late 1950s to provide a relatively machine-independent language for solving problems in the field of business. Although its applications are virtually nonexistent outside the business community, it nevertheless ranks close to Fortran in percentage use.

Basic (Beginners All-purpose Symbolic Instruction Code). In the early 1960s, a project was undertaken at Dartmouth College to make the computer more accessible and easy to use by both students and faculty. One of the objectives was to develop a language that could be quickly learned but powerful enough to be used in solving most small- to medium-scale problems in any discipline. The result of this effort was Basic, and it more than met its objectives.

PL/I (Programming Language One). Developed by IBM for its System/360 series of computers, this language was designed to serve the needs of both the scientific and the business communities. Currently, PL/I is the most powerful language available on a widespread basis, and its use certainly leads to more efficiency. The use of PL/I is still relatively small, however, apparently because Fortran and Cobol meet the needs of their respective users reasonably well. While the use of PL/I should continue to grow, it does not appear that its use will surpass that of Fortran or Cobol in the near future, and maybe never.

Algol (Algorithmic Language). Being particularly oriented toward ease of expression of algorithms, Algol has enjoyed widespread use as a standardized language in which algorithms can be expressed in the published literature. Ordinary mathematical expressions and English words are incorporated in such a manner that it is a powerful language yet simple to use. Oriented toward the scientific community, Algol competes directly with Fortran, but the latter certainly dominates in this country. One reason is that implementations of Algol have generally fallen short of the published standards for the language. Another is that in the late 1950s, Fortran was enthusiastically promoted by IBM, which dominates the computer market in this country. Algol has fared much better in the European market, which is not so dominated by IBM.

APL (A Programming Language). Although not widely used, APL directly incorporates a rather large number of commonly used mathematical operations, which make the language very useful in the scientific community. Unlike PL/I, it has very little appeal to the business community. Many mathematical problems can be more readily expressed in APL than in any of the other languages, which should certainly promote its use.

Numerous other languages are available on a limited basis, that is, limited to a few installations or to machines of a single manufacturer. As programming languages continue to evolve, new languages will appear and significant improvements will be made in existing ones.

While the above discussion might lead one to believe that all students in an introductory course will be studying Basic, this assumption is certainly not the case. The specific language studied will depend upon a number of factors, including what languages are available on the computer at the computing center, the burden imposed upon the center by use of a specific language, and the language requirements of other courses in the curriculum.

As our next topic, we shall program our salesman's-pay problem in two of these languages, namely, Fortran and Basic. Except for Cobol (which is not very convenient for small problems of the type considered in this text), these two languages are undoubtedly the most commonly used. In reading these next two sections, the student should consider that *the entire program is first read by the computer, which then begins to perform the indicated operations*. We shall examine this aspect in more detail in a subsequent section of this chapter.

1.5. THE BASIC PROGRAMMING LANGUAGE

As Basic is certainly one of the simplest programming languages to learn, we shall begin with it. Figure 1-3 presents the Basic program corresponding to the flowchart in Figure 1-2. As illustrated in Figure 1-3, a Basic program consists of a sequence of statements. Each line in the program begins with a *line number* and contains a single statement. Beginning with the first statement in the pro-

```

10 INPUT B,S
20 IF S>100 THEN 50
30 LET C=0
40 GO TO 60
50 LET C=0.04*(S-100)
60 LET P=B+C
70 PRINT "TOTAL PAY IS" P
80 END

```

a. Program

```

?
82.50,476.13

```

b. Input

```

TOTAL PAY IS 97.545

```

c. Output

Figure 1-3 Basic program for salesman's pay problem

gram, the computer executes (that is, performs the indicated operations) the statements one at a time. Upon completion of the operations entailed in executing one statement, the computer proceeds to the statement with the next highest line number *unless* directed to do otherwise as a result of the operations performed during the execution of the current statement. Actually, the lines in a Basic program can be entered in any order, but an ascending order with respect to the line numbers as in Figure 1-3 makes analysis of the program much easier.

To illustrate this in more detail, let's analyze the statement in line 20 of the Basic program in Figure 1-3:

20 IF S > 100 THEN 50

In this case, the statement to be executed upon completion of the statement in line 20 depends upon the current value of S, which was specified by the INPUT statement in line 10. If the value of S exceeds 100, the statement in line 20 directs the computer to execute the statement in line 50. If the value of S does not exceed 100, the computer proceeds to the statement with the next highest line number, namely, 30.

To understand the program in Figure 1-3, perhaps the best approach is to perform a "trace" of the program, that is, execute the statements in the same manner as would the computer. We shall assume that the programmer is communicating with the computer via a teletype or typewriter. He first enters the entire program in Figure 1-3a and then instructs the computer to "run," or execute, his program. The computer begins with the statement with the lowest line number:

Line 10. Numerical values are to be entered for B and S. Upon execution of this statement, the computer prints a ? character on the teletype or typewriter, and the programmer then responds with the numerical values of B and S, separated by commas. This step is indicated in Figure 1-3b. After this statement has been executed, the values of B and S are 82.50 and 476.13, respectively.

Line 20. Since the current value of S (namely, 476.13) exceeds 100, the next statement to be executed is the one in line 50.

Line 50. Using the current value of 476.13 for S, we have the value of C computed in line 50 as

$$C = 0.04 * (476.13 - 100) = 15.045$$

The statements follow the normal rules of algebra with the asterisk (*) representing multiplication.

Line 60. The value of P is computed as the sum of the current values of B and C, namely, 82.50 and 15.045, respectively:

$$P = 82.50 + 15.045 = 97.545$$

Line 70. The characters TOTAL PAY IS followed by the current numer-

ical value for P are printed on the output printer or typer. This output is shown in Figure 1-3c.

Line 80. This statement informs the computer that all operations required by this program have been completed.

More details on the Basic programming language are available in Appendix B. This example by no means illustrates all the features available in Basic, but the simplicity and straightforward nature is retained throughout other features of the language.

1.6. THE FORTRAN PROGRAMMING LANGUAGE

The Fortran program corresponding to the flowchart in Figure 1-2 is presented in Figure 1-4. In Fortran the program consists of a sequence of statements, each generally appearing on a separate line. At the programmer's discretion, a statement may or may not be given a statement number. In the program in Figure 1-4, only two of the nine statements are given statement numbers.

Execution of a Fortran program begins with the first statement in the program and then proceeds statement by statement in the order that they appear *unless* execution of the current statement directs the computer to do otherwise. To enhance understanding of the program in Figure 1-4, we shall again "trace" the execution of the statements, beginning with the first statement:

READ,B,S

This statement instructs the computer to obtain numerical values for B and S . When we submit a program in the form of punched cards, these values are generally obtained from a "data card" entered immediately following the For-

```

READ,B,S
IF(S.GT.100.) GO TO 10
C=7
GO TO 20
10 C=0.04*(S-100.)
20 P=B+C
PRINT,'TOTAL PAY IS',P
STOP
END

```

a. Program (in all printouts of Fortran programs in this text, apostrophes are used instead of quotation marks)

82,50 476,13

b. Data card

TOTAL PAY IS 0.9754518E 02

c. Output

Figure 1-4 Fortran program for salesman's pay problem

INTRODUCTION

16

tran program. For this example, the values entered are given in Figure 1-4b. After execution of this READ statement, the values of B and S are 82.50 and 476.13, respectively.

IF(S.GT.100.)GOTO10

In this statement, the .GT. means “is greater than.” This statement directs the computer to execute statement 10 if the value of S exceeds 100; otherwise, the next statement to be executed would be

$$C = 0$$

which immediately follows. In our case, the numerical value of S is 476.13. Since this exceeds 100, the next statement to be executed is statement 10.

10 C = 0.04* (S - 100.)

In this statement the value of C is computed as follows (the current value of S is 476.13):

$$C = 0.04 * (476.13 - 100.) = 15.045$$

The * is used to denote multiplication.

20 P = B + C

This statement instructs the computer to add the current values of B and C to obtain a value for P. Since the current values of B and C are 82.50 and 15.045, respectively, the result is

$$P = 82.50 + 15.045 = 97.545$$

PRINT, “TOTAL PAY IS”, P

This statement instructs the computer to print the characters TOTAL PAY IS followed by the numerical value of P. As illustrated in Figure 1-4c, the numerical value of P is expressed in the somewhat unusual form 0.9754518E 02, which stands for the scientific, or exponential, notation

$$0.9754518 \times 10^2$$

In this notation, the exponent of 10 effectively shifts the decimal place to the left or right. In our case, 10^2 equals 100, which causes the decimal point to be shifted two places to the right, yielding 97.54518. In performing numerical computations, Fortran carries about seven digits, all of which have been printed in this example.

STOP

This statement informs the computer that all operations called for in this problem have been completed. No further statements in this particular program will be executed.

Although the Fortran program in Figure 1-4 is quite similar to the Basic program in Figure 1-3, the languages are significantly different. Fortran has numerous additional features, most of which are substantially different from the corresponding features in Basic if, in fact, they exist in Basic at all. A more detailed description of the Fortran language is presented in Appendix C.

1.7. MODEL OF A SIMPLE COMPUTING MACHINE

Using a computer is somewhat analogous to driving a car in that the user need not be thoroughly knowledgeable about how it works. What percentage of automobile drivers could tell you what goes on during the four cycles of the internal combustion engine found in virtually every automobile? Of these, how many could tell you how the automatic transmission works? If they cannot, does this mean that they are unsafe drivers and should not be given a driver's license? Of course not.

In the same fashion, many excellent computer programmers cannot draw the diagram of the circuit that adds two numbers. Many applications programmers in the business and scientific areas are not fluent in the use of the binary number system, which is the system in which most modern computers basically operate.

Although one does not need to know very much about an automobile to drive it, some knowledge can lead to more effective or pleasant use of the vehicle. For example, low tire pressure gives a soft ride but a higher gas consumption and shorter tire life. Slightly higher tire pressure gives a slightly less comfortable ride but better gas mileage and tire life.

In the next few sections, we shall present a simplistic model of a computer so that the user may get some idea of how it works. The objective is to give the reader some insight into the principles involved and thus shall in certain cases make things a bit simpler than they really are. In a later chapter, we shall analyze in more detail the elements that comprise a computing system.

The model of a computing machine illustrated in Figure 1-5 is quite representative of many actual computers. The heart of this device is the Central Processing Unit (or CPU for short). This unit is responsible for the general supervision of operations that are going on inside the computer. A section, or subunit, of the CPU is the arithmetic unit, which is basically capable of the following operations:

1. Adding one number to another.
2. Subtracting one number from another.
3. Multiplying one number by another.
4. Dividing one number by another.
5. Ascertaining whether a number is positive, negative, or zero.

Few arithmetic units are capable of directly taking the square root of a number. Operations such as this are accomplished by sequences of the five operations given above. We shall examine several such procedures in later sections of this text.

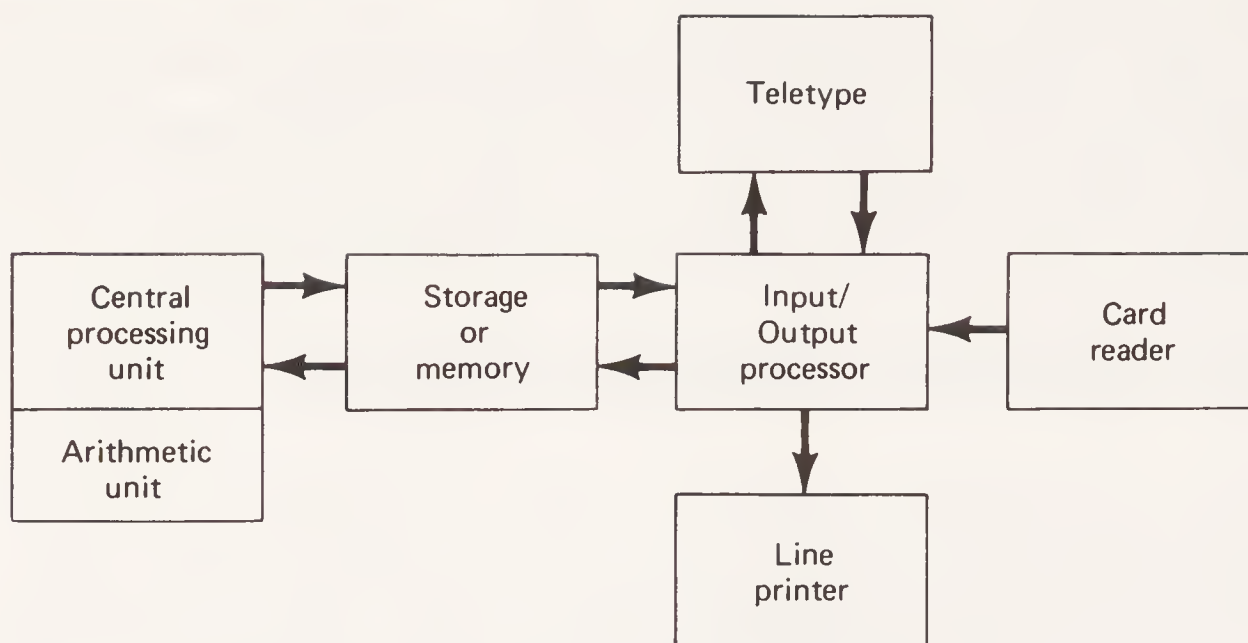


Figure 1-5 Schematic model of a computing machine

The CPU is capable of performing many operations itself, such as the manipulation of characters. For example, it can compare two characters for equality in much the same fashion that it compares two numerical values. It also can perform the operations necessary to add, delete, or modify characters appearing within a string of characters.

The Input/Output Processor (or IOP for short) communicates with the actual input/output (or I/O for short) devices. For example, suppose the computer is to read a number from a card. The CPU then instructs the IOP that a card is to be read. The IOP in turn “calls up” the card reader and asks for the information on the next card. After relaying this information to the IOP, the card reader then “throws away” the card. The IOP inserts the information on the card into the appropriate area in memory, from which the CPU can access it.

Information can be stored and retrieved from memory by the CPU. In actual computers, the memory is divided into elements called *words*. Each word is given a unique “address.” The CPU then directs memory to store an item of information at a certain address. The address itself is not stored; only the item of information is stored. Similarly, the CPU can retrieve from memory whatever information is stored in the word located at a certain address.

While memory basically serves a single function as a store for information, the information contained therein can be classified into two categories. The first of these is the information such as numerical values that the computer is to use in performing the desired computations. This type of information is referred to as *data*. The second category is the information specifying the sequence of operations, that is, the *instructions*, that the computer is to perform in executing a program. For our purposes, we might envision one area of memory as containing the instructions and another area as containing the data. The fact that in practice these two types of information are not located in separately designated areas but interspersed throughout memory is immaterial to the present discussion.

To illustrate the operations that proceed, suppose we examine the computa-

tion of adding two numbers stored as data in memory to obtain a third number that is to be stored. In performing its operations, that is, executing its instructions, the central processing unit utilizes one or more memory elements (implemented within the CPU) called *accumulators* through which it performs its operations. For our example of adding two numbers, the following instructions will be needed:

1. *Load Word* (LW). Copy data currently stored in a location in memory into one of the accumulators.
2. *Add Word* (AW). Add the data (a numerical value) currently stored in a location in memory to the data (a numerical value) currently in one of the accumulators.
3. *Store Word* (STW). Copy the data currently in one of the accumulators to a location in core memory.

The letters in parentheses following the above instructions constitute an abbreviation known in computer science as a *mnemonic*.

The following representation of the three instructions to add our two numbers reflects the information contained in the area of memory reserved for the instructions:

LW,3, Address of the storage location for the first number

AW,3, Address of the storage location for the second number

STW,3, Address of the storage location at which result is to be stored

The “3” following each mnemonic designates that the third accumulator is to be used in executing these instructions.

Each of these instructions would occupy a single word of the memory. The CPU retrieves the first instruction from memory, performs the designated operation, and proceeds to the next. The use of memory in this fashion was basically the stored-program concept of John von Neumann. In executing these instructions, the computer uses data stored in other areas of memory.

Specifically note the use of the word “copy” in the explanation of the store-word and load-word instructions given previously. The load-word instruction *copies* the information in a word of memory into the accumulator without altering in any way the information stored in that word of memory. Similarly, the store-word instruction *copies* the contents of an accumulator into a word in memory and does not alter the information in that accumulator. Whatever information, however, was previously stored in this word in memory, has been destroyed. Although not specifically noted, the add-word instruction adds the value stored in a word of memory to an accumulator and does not alter the information stored in the word of memory.

A more direct expression of the concept discussed in the above paragraph is that a READ operation on memory is nondestructive (that is, the stored information is not altered), whereas a WRITE (or STORE) operation on memory is

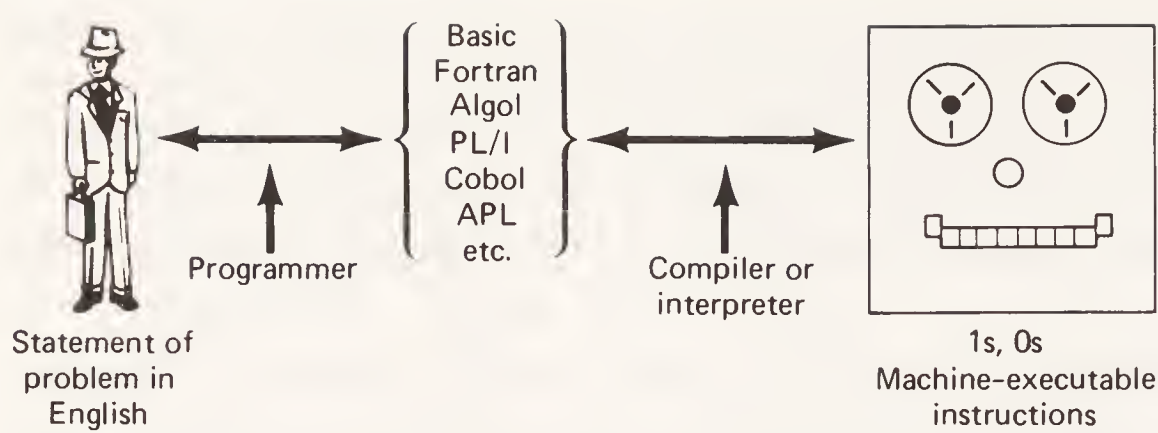


Figure 1-6 The role of programming languages

destructive in that the information previously stored therein is no longer available.

In a previous section, programming languages such as Basic or Fortran were presented as a vehicle by which a programmer could communicate to the computer the operations it should follow in solving his problem. While both the Basic program in Figure 1-3 and the Fortran program in Figure 1-4 contain a statement calling for the addition of two numbers, their structure is not very similar to the machine instructions described in this section.

Just as the problem solver must translate his problem from its original statement in English to a sequence of statements in a programming language, another translation must be made from the statements in the programming language to a sequence of instructions that can be directly executed by the computer. Figure 1-6 illustrates this situation. This second translation is performed by the computer by means of either a *compiler* or an *interpreter*, which we shall examine in the next section. While it is possible for the programmer to create a program directly in machine-executable instructions, this does not generally lead to the most productive problem solving.

1.8. INTERPRETERS AND COMPILERS

While any programming language could be implemented via an interpreter, this approach has been very commonly used in implementing Basic and APL. On the other hand, Fortran, Cobol, Algol, and PL/I are generally implemented via a compiler. In this section, we want to explain how this is done.

In essence, an interpreter is a program that treats the statements in the programming language as data. In processing the program, the interpreter reads the statements in the programming language into memory. When executing the program, the interpreter scans through the statements, decides what operations to perform, and performs them. As illustrated in Figure 1-7, both the interpreter and the user's program are stored in memory.

Since the interpreter is written in machine-executable instructions, the CPU is actually executing instructions only from within the interpreter itself. By executing these instructions within the interpreter, however, the CPU effectively decodes the statements in the programming language. Depending upon the

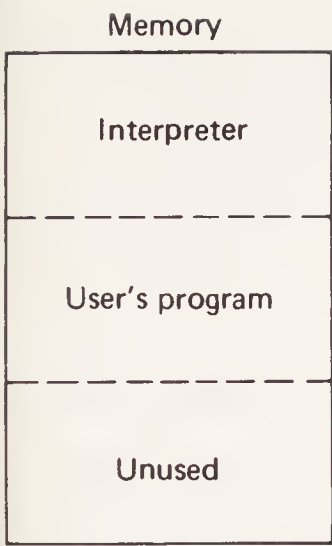


Figure 1-7 Allocation of memory for implementation of a programming language via an interpreter

nature of the statement, different instructions within the interpreter are executed in order to effectively perform the operations specified by the programming language itself.

Compilers basically translate the statements in the programming language into an equivalent set of machine-executable instructions. Since this must occur prior to the performance of any operations specified within the program, this act of translation is generally referred to as the *compilation phase*.

Compilers come in two “styles,” one of which is referred to as a “load-and-go” compiler. In this system, the compiler reads the statements in the program, generates the corresponding machine-executable instructions, and places them directly into another area of memory (see Figure 1-8). During this process, the CPU is executing the set of instructions that comprise the compiler. When the last statement in the program has been processed, a complete set of machine-executable instructions is stored in another area of memory, and the compilation phase is completed. An instruction in the load-and-go compiler then directs the CPU to the very first instruction in the set of instructions generated from the user’s program, and the *execution phase* begins. The CPU executes this set of instructions until all operations called for by the user’s program have been completed. For any logical termination point in the user’s program (such as the

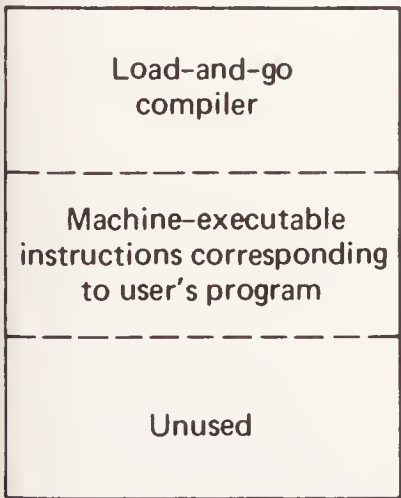


Figure 1-8 Memory allocation for a load-and-go compiler

STOP statement in the Fortran program in Figure 1-4), the load-and-go compiler generates instructions that direct the CPU to return to a predetermined location within the compiler itself. Thus, when these instructions are executed during the execution phase, the CPU is directed to return to the compiler, thus terminating the execution phase. At this point, the user's program has been completed, and the compiler can instruct the computer to proceed to the next program to be run.

Instead of storing the generated instructions directly into memory, many compilers produce the set of machine-executable instructions on punched cards, magnetic tape, or other medium suitable for subsequent re-entry of the information into the computer. The sequence of phases is as follows (refer to Figure 1-9):

- 1. *Compilation Phase.* The compiler is entered into memory, followed by the statements written in the programming language. (This program is called the *source program*.) The compiler generates the machine-executable instructions on some medium from which they can be subsequently re-entered into the computer. This output is generally referred to as the *object deck*.
- 2. *Load Phase.* Before the set of instructions can be executed, they must be entered into memory. The *loader* is a small program that reads the

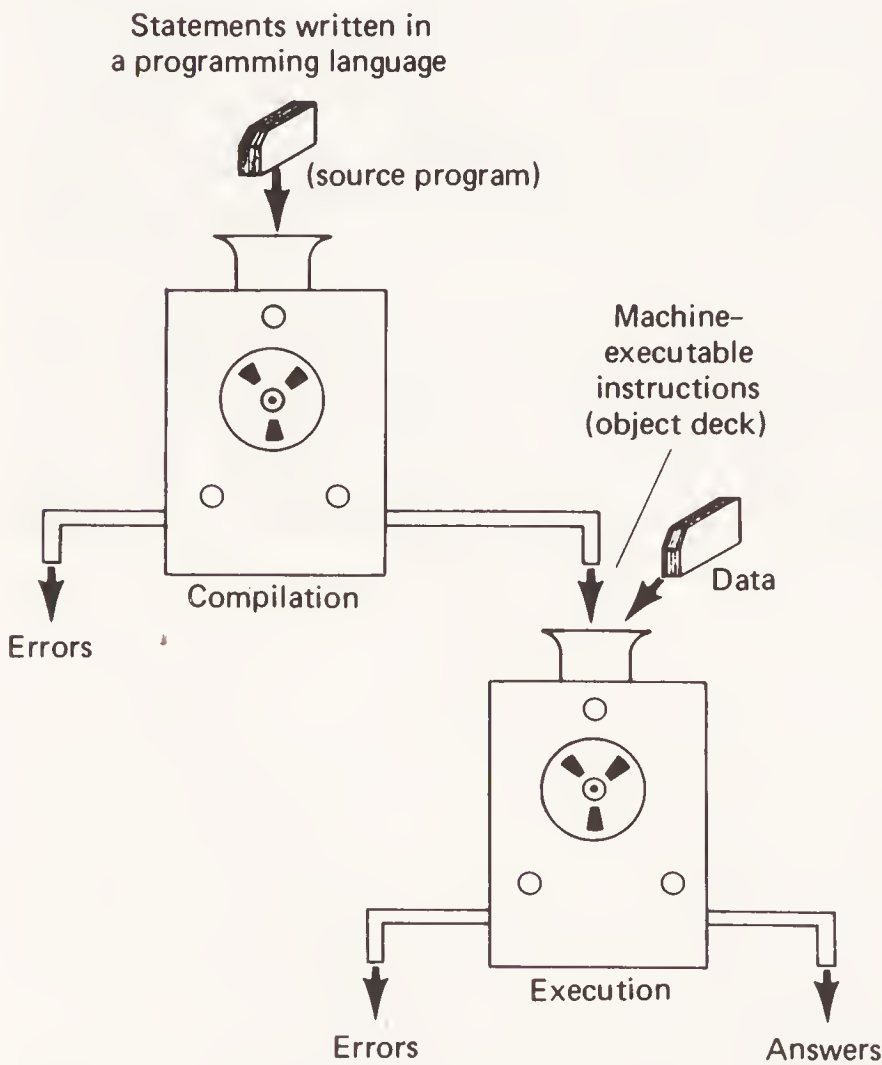


Figure 1-9 Schematic representation of compilation and execution phases for a compiler that generates an object deck

instructions and places them into memory. Since the compiler's task has been completed, these instructions can be placed in the same area of memory that was used for the compiler.

3. *Execution Phase.* Upon completion of the load phase, the loader directs the CPU to the first instruction in the user's program.

As for the relative advantages of the two types of compilers as compared with an interpreter, the following observations are pertinent:

1. Although we will not consider errors in detail until the next section, the interpreter is generally able to provide the programmer more information concerning his error, since a copy of the program is stored in memory. Load-and-go compilers also have been quite good in this respect, with compilers that produce object decks being poorest of all.
2. As we shall see in a later chapter, programmers frequently direct the computer to execute the same statement or set of statements a number of times within the same run. Since the interpreter must decode the statement again each time it is executed, some "repeat work" is entailed. Compilers decode all statements once, regardless of how many times they are executed.
3. The load-and-go compiler and the interpreter reside in memory during the execution of the program. Compilers that produce an object deck need not reside in memory past the compilation phase, thus freeing this memory for use by the program. Thus, larger programs can be run.
4. If the same program is to be executed several times, an object deck can be saved and the compilation phase avoided in all except the first run. Since the load-and-go compiler produces no object deck, the compilation phase must be repeated. Compile time for load-and-go compilers is generally short, however.

Although a number of other observations could be made, the general situation is that for small- to medium-scale programs, an interpreter or a load-and-go compiler has definite advantages, particularly with respect to the error messages generated. Large users or programmers in production-oriented centers prefer compilers that produce an object deck.

1.9. DEBUGGING

Except for a few brief references in the previous sections, we have avoided any discussion of programming errors up to this point. The term *bug* is commonly used as a connotation for an erroneous condition within a program, and the act of removing these, that is, correcting the program, is referred to as *debugging*.

The possibility of the existence of errors necessitates a revision of the problem-solving procedure presented in Section 1.2. In fact, the flowchart in Figure 1.10 is probably more representative of the normal approach to solving a

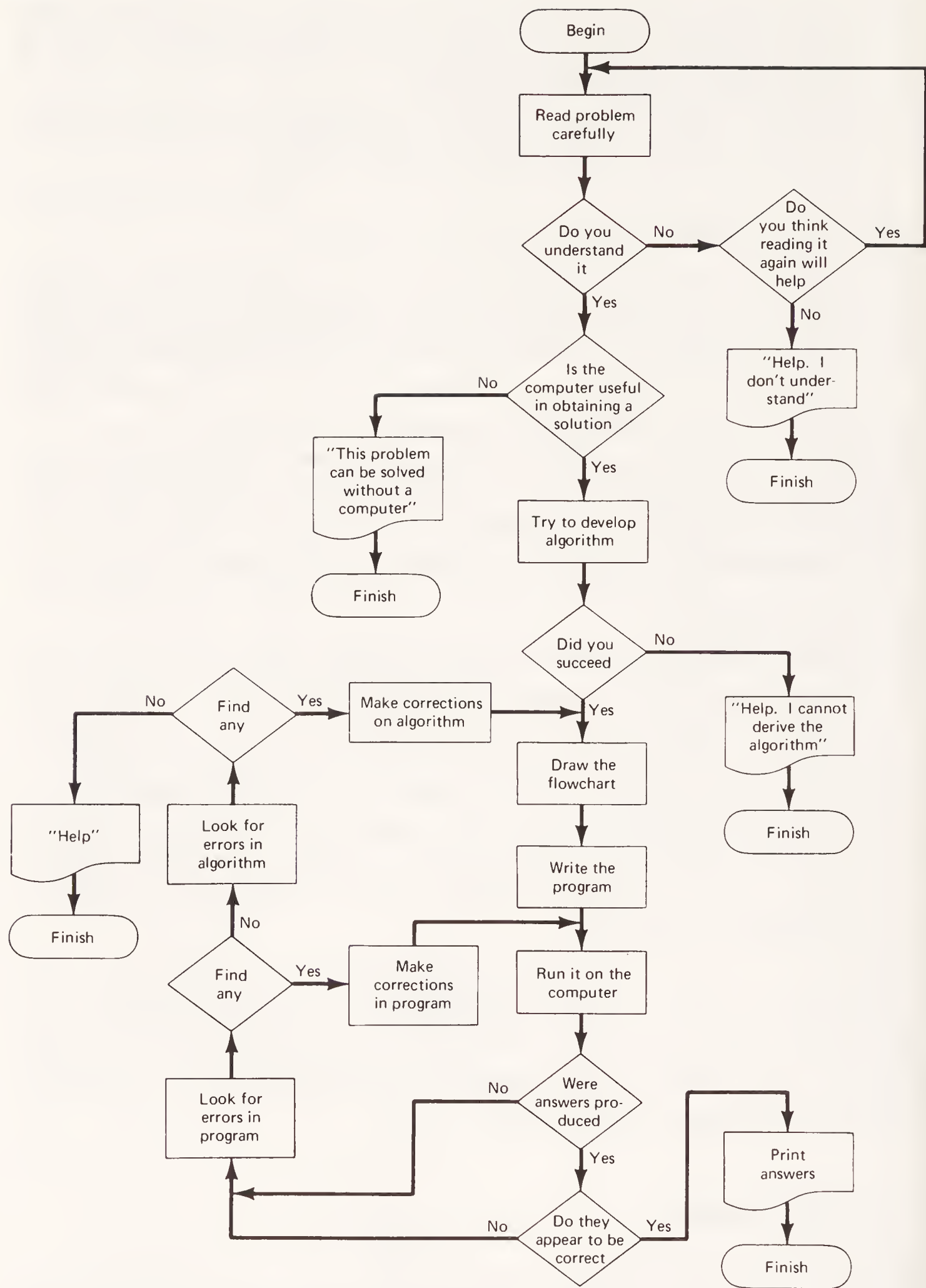


Figure 1-10 Flowchart of problem-solving procedure

problem. In many cases, the identification and removal of bugs can be more time-consuming than developing the original program.

From a single processing (or “run”) of the program, errors of two different types may be generated:

1. A *compilation error* means that the compiler has detected a situation in which the programmer has failed to follow the rules of the language. For example, neither Fortran nor Basic would accept the following construction:

$$0.04(S - 100)$$

The asterisk denoting the multiplication operation has been omitted. Errors of this type are generally easy to remove.

2. An *execution error* means that an abnormal situation has been encountered during execution of the program. Examples of an abnormal condition could be attempted division by zero, attempting to take the square root of a negative number, or other similar situations. These could mean problems in either the program (that is, algorithm was not programmed correctly) or in the algorithm (proposed algorithm is not correct).

In other cases, no errors will be produced, but the answers are incorrect. Again, this may be due to an error made during programming or to an error made during developing the algorithm. For example, changing the statement

$$\text{LET } C = 0.04 * (S - 100)$$

to

$$\text{LET } C = 0.04 + (S - 100)$$

will result in a program that will produce an answer without any error messages being generated. The programmer instructed the computer to add instead of multiply, which can certainly be performed. The computer has no way of detecting that multiplication instead of addition should be performed.

In still other cases, the program will produce correct answers in some cases but incorrect answers in others. Suppose that the statements in lines 20 through 40 are omitted from the Basic program in Figure 1-3, yielding

```
10 INPUT B,S
50 LET C = 0.04*(S - 100)
60 LET P = B + C
70 PRINT "TOTAL PAY IS" P
80 END
```

This program will produce correct answers as long as the value of S exceeds 100. Otherwise, the program produces incorrect answers but without any error messages being generated.

Students should be cautioned to be on the lookout for bugs of the last two types. In general, a case for which the solution is known should be run first to verify that the program is correct for at least one case. For programs with decision blocks, enough runs should be made so that all statements within the program are executed at least once, and the answers produced should be verified

26 for correctness. While even these precautions unfortunately do not guarantee that the program is correct, they will detect most bugs (but by no means all).

1.10. BATCH-PROCESSING SYSTEMS

With respect to orientation toward users, computing systems can be divided into two broad categories: batch-processing systems versus conversational-time-sharing systems. Since batch-processing appeared first chronologically, we shall consider it before considering conversational time-sharing.

As the operational complexity of general-purpose computing systems continued to increase, centers began to use professional computer operators to operate the machines instead of permitting the individual programmers to operate them. Furthermore, the load at most centralized facilities is such that a number of jobs are almost always waiting to be run. Thus, the programmer brings his program to the center, leaves it to be run by the professional operators, and returns for it at some later time. The term *turnaround* encompasses these steps. How long he has to wait (referred to as the *turnaround time*) depends upon a number of factors: the current load on the center, the priority of his work, the running time and memory requirements of his program, etc. The number of times he can have his program run during a day is determined by the turnaround time. Centers at which the programmers are limited to four turnarounds or fewer per day are not unusual, although many centers do considerably better.

In explaining the batch-processing mode of operation, we shall use the system in Figure 1-11 as the basis of our discussion. In reality, this is a relatively minimal configuration but will suffice for our purposes.

In the system of Figure 1-11, programs are submitted to the system in the form of punched cards. As illustrated in Figure 1-12, a punched card is divided

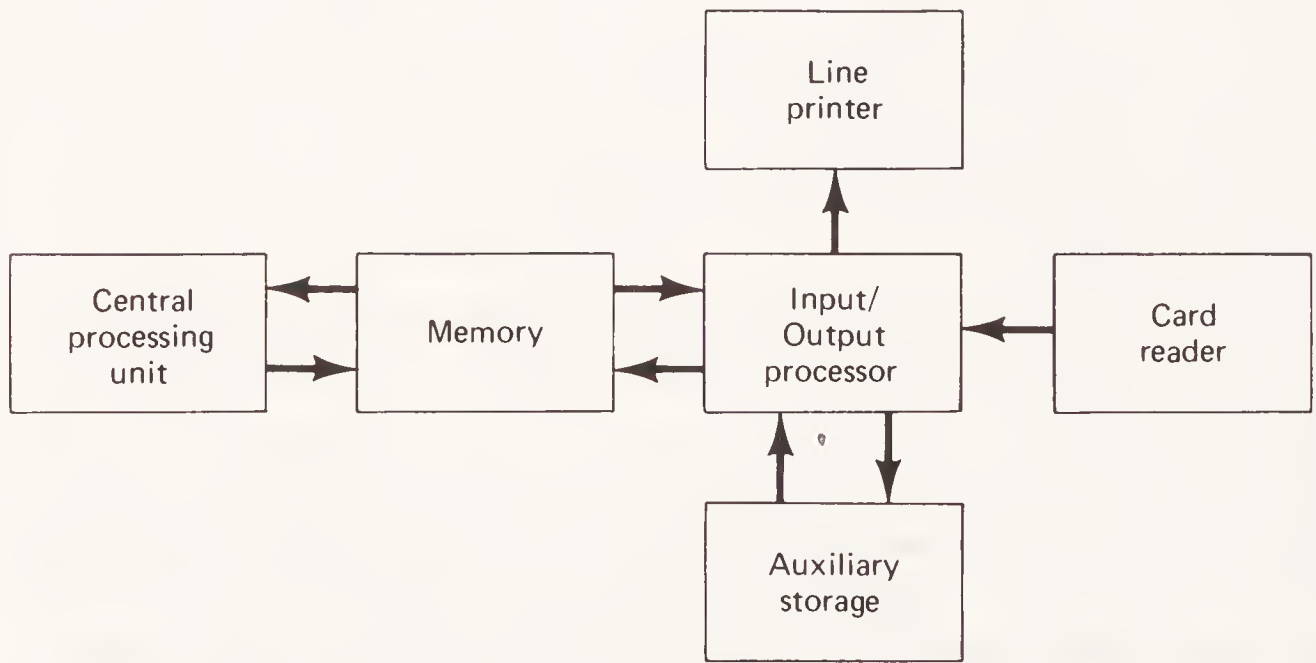


Figure 1-11 System configuration for batch processing

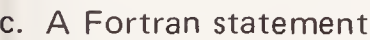
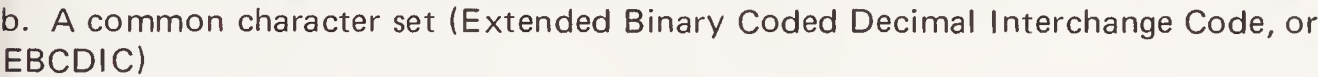


Figure 1-12 Examples of punched cards

into 80 columns, each of which may contain a single character. Within a single column, there are twelve possible locations at which a hole may be punched. For a specific character code, such as EBCDIC, each character is represented by a unique combination of hole punches. The EBCDIC character code in Figure 1-12b is probably the most commonly used but is by no means universal. Statements in the program are punched onto the cards, as illustrated for a Fortran statement in Figure 1-12c.

The holes are punched into the cards by keypunch machines such as the one illustrated in Figure 1-13a. The keyboard on this machine is similar to that on the common typewriter except that all letters are uppercase and that the digits and special symbols (parentheses, slash, etc.) are arranged differently. Large batch centers usually have one or more rooms such as the one illustrated in Figure 1-13b where programmers have access to keypunch machines.

In a batch-processing environment, the programs are submitted along with any necessary *control cards* that inform the system as to what actions it is to perform. Figure 1-14 illustrates a typical deck complete with control cards for processing a Fortran program. In computer terminology, this complete deck is called a *job*.

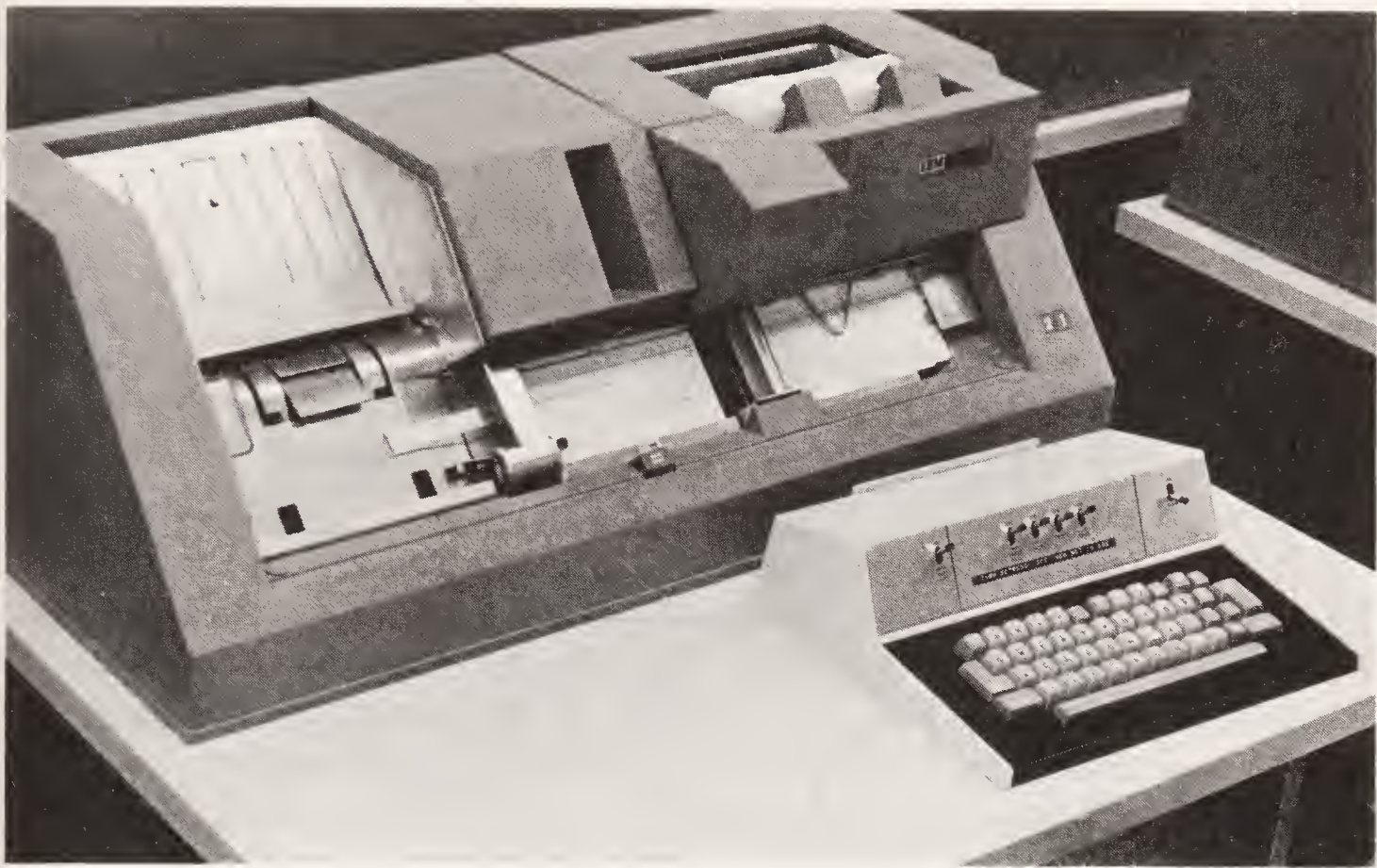
In a batch-processing environment, a supervisory-type program often referred to as the *monitor*, *executive*, or *operating system* is responsible for directing the computing system through whatever sequence of events is necessary to process the job. The control cards basically provide a mechanism by which the programmer informs the operating system as to the nature of his job and what actions will be necessary to process it.

In the deck illustrated in Figure 1-14, each control card begins with the character \$. The \$JOB control card indicates the beginning of a new job. The programmer's name, his identification number for bookkeeping purposes, an estimated maximum run time (1 minute), and an estimated maximum on the pages of output (twenty pages) are given. When the run time reaches one minute or the printed output reaches twenty pages, the job will be "aborted," that is, terminated, and the next job begun.

The next control card, \$FORTRAN, indicates that the programmer is submitting a Fortran program. In response to this card, the operating system directs the CPU to copy the Fortran compiler from the auxiliary storage device into memory. Once this is completed, the statements comprising the Fortran program are entered. The LIST option on the \$FORTRAN card instructs the compiler to list the source program as it is being compiled. Assuming that the compiler produces an object deck, it would be written into a preassigned area on the auxiliary storage unit.

Upon encounter of the \$LOAD card, the operating system copies the loader from the auxiliary storage device into memory. The loader in turn copies the object deck from the auxiliary storage device into memory. Of course, if a load-and-go compiler or an interpreter is used, this step will be unnecessary.

Upon processing the \$EXECUTE card, the operating system directs the CPU to the first instruction in the user's program. If the Fortran program contains



a. Keypunch machine



b. Input/Output room where programmers correct their programs, submit jobs, and pick up returned jobs

Figure 1-13 Keypunch facilities for preparing card decks (Photos courtesy Louisiana State University)

any READ statements, the actual READ operation does not occur until the program is being executed. The program in Figure 1-14 contains one READ statement. Therefore, one card containing the two numerical values called for is placed immediately after the \$EXECUTE control card.



Figure 1-14 Typical deck for batch processing

Upon completion of the job, the operating system proceeds to the next control card to ascertain what other actions are needed. The \$FINISH card informs it that nothing else is to be done. Were another job waiting to be run, the \$FINISH card would be removed and the next job immediately entered into the card-reader.

While the batch-processing system described in this section is fairly representative of one used on small- to medium-size computers, larger computers provide much more flexibility and capability in manipulating the processing of jobs to achieve maximum efficiency. The basic process, however, at least as far as the user is concerned, is a little different from that presented here.

Depending upon the user, he might find this mode of operation very convenient or very inconvenient. Anyone who is running a large problem or one that runs a long time is generally content to leave his job for someone else to run when his turn comes. For the small user whose program is very short and runs quickly, the waiting time between job submission and return is, to say the least, inconvenient. Conversational time-sharing is a more attractive alternative.

During the debugging process, the batch-processing approach can be especially inconvenient. For example, suppose the first run of the program gives the results in Figure 1-15a. Removing the extra parenthesis gives the results in Figure 1-15b. Although no syntax error exists, this program produced the obviously incorrect answer of \$458.67. In the computation of C in statement number 10, a + has been punched instead of an *. Correcting this error produces the correct program in Figure 1-15c. Since three runs are needed, the elapsed time is about three times the turnaround time.

1.11. CONVERSATIONAL TIME-SHARING

The basic idea behind conversational time-sharing is that several users have programs in progress simultaneously on the same computing system. In the simplest implementations, the only components needed in the computing system are the CPU, some memory, and a teletype for each of the users. Figure 1-15a illustrates the configuration of a conversational-time-sharing system to support four users simultaneously. Such small systems are generally dedicated to a single programming language, usually Basic. As illustrated in Figure 1-16b, the Basic interpreter occupies a region of memory, and the remainder is divided between the four users of the system.

In addition to receiving his proportionate share of the system's memory, each user gets his proportionate share of the processor's time. The processor's time is divided into small increments called *slices*. A user receives the processor's undivided attention for one time slice but then receives no attention over the next three time slices. Therefore, he is receiving 25 percent of the processor's time less the small overhead entailed in switching from one user to the next. These time slices are relatively small, so that each user feels that he has the

```

1          READ,B,S
2          IF(S.GT.100.) GO TO 10
3          C=0
4          GO TO 20
5          10 C=0.04+(S-100.)
6          20 P=B+C)
***ERROR*** UNMATCHED PARENTHESIS
7          PRINT,'TOTAL PAY IS',P
8          STOP
9          END

```

a. Compile-time error

```

1          READ,B,S
2          IF(S.GT.100.) GO TO 10
3          C=0
4          GO TO 20
5          10 C=0.04+(S-100.)
6          20 P=B+C
7          PRINT,'TOTAL PAY IS',P
8          STOP
9          END

```

b. Error in programming logic

```

1          READ,B,S
2          IF(S.GT.100.) GO TO 10
3          C=0
4          GO TO 20
5          10 C=0.04*(S-100.)
6          20 P=B+C
7          PRINT,'TOTAL PAY IS',P
8          STOP
9          END

```

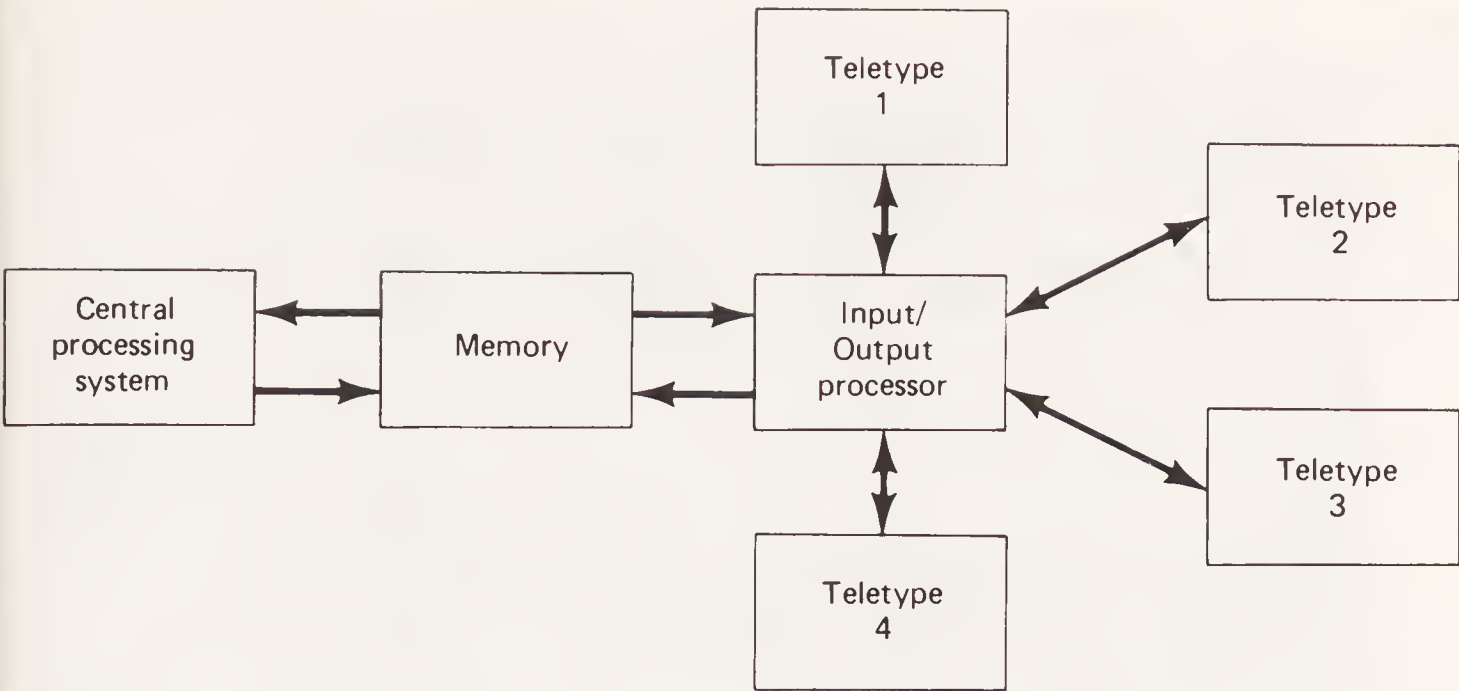
c. Correct program

Figure 1-15 Debugging a program in the batch mode

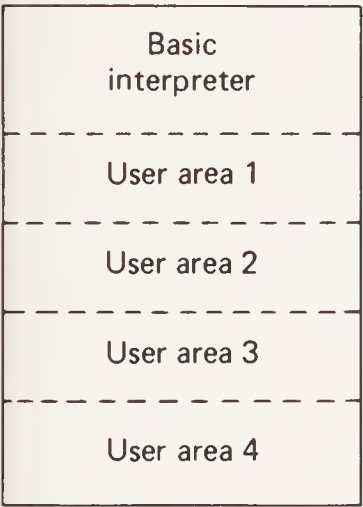
computer's continuous services. Although his program runs slower (as measured by the "clock on the wall") than if he were the only user, this is more than offset by the added convenience of being able to get his fraction of the machine's services immediately rather than waiting to have the entire computer at his service.

Although conversational-time-sharing systems are commercially available in configurations similar to the one in Figure 1-16, machines with larger configurations can offer a wider spectrum of services. Large conversational-time-sharing systems can simultaneously serve fifty or more users. If an auxiliary storage unit is available, each user can be assigned some space in which he can store programs or data to be available the next time he uses the system. In addition to Basic, large systems generally offer other programming languages such as Fortran.

In the early time-sharing systems, users communicated with the computer with teletypes such as the one illustrated in Figure 1-17a. Although these units operated at the relatively slow speed of 10 characters per second, they neverthe-



a. Configuration



b. Memory allocation

Figure 1-16 Small conversational time-sharing system dedicated to Basic

less proved to be satisfactory. More recent models operate at 30 characters per second and are usually much quieter than their predecessors. Many systems now use the cathode-ray-tube unit illustrated in Figure 1-17b.

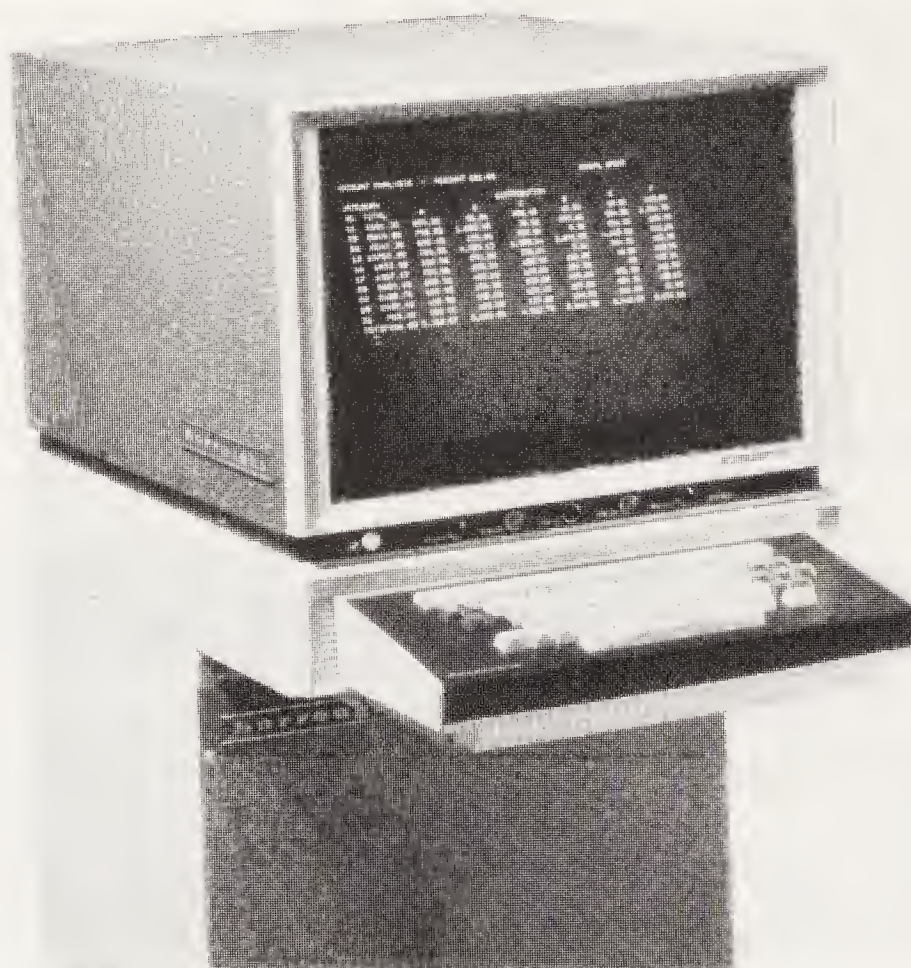
In many systems, the communication between terminal and computer is over telephone lines, which permits the user to be located hundreds of miles from the computer. When he wants to use the computer, he dials its number in the same manner that he places other calls. At the center, units such as those illustrated in Figure 1-18 answer with a characteristic “beep.”

To illustrate the conversational nature of the dialogue between the user and the computer, a typical teletype printout is shown in Figure 1-19. All computer responses are underlined. In the left margin, numbers have been added to facilitate explanation of the printout.

- 1. Upon completion of the call to the computer, it responds with a message identifying the system along with other data of general interest such as the time of day and the number of other users.



a. Teletype (Courtesy Control Data Corporation)



b. Cathode-ray tube unit (Courtesy Burroughs Corporation)

Figure 1-17 Terminals

2. The user must enter his identification number for bookkeeping purposes. To prevent the unauthorized use of this number by others, a password must be entered but is not printed on the teletype output. A proper ID number followed by an improper password will not be accepted as valid.
3. The programmer informed the computer that he wants to run a Basic program.



Figure 1-18 Phone units located at the computer center that provide “dial-up” service in time-sharing systems (Photo courtesy Louisiana State University)

4. By entering NEW, the user indicated that the program was to be created at the terminal. The first file name, JOHN, that was entered was not accepted because the user already had a file by that name in the system. The name MARY was acceptable. Upon completion of the necessary housekeeping chores associated with creating a new file, the computer types READY to inform the user that he can proceed.
5. The Basic program is entered from the terminal. Many systems scan the statements for syntax errors as they are entered, thus indicating errors immediately so that they can be corrected before proceeding. Other systems, of which ours is an example, do not scan for errors until an attempt is made to execute the program.
6. By entering SAVE the programmer instructed the computer to store a copy of the program on the auxiliary storage unit. Upon completion, the computer again types READY.

```

1 { HILLBILLY TIME-SHARING SERVICE
   ON AT 10:27 TTY: 07

2 { ENTER USER NO: X547
   ENTER PASSWORD:

3 { SYSTEM? BASIC

   OLD OR NEW? NEW
   NEW FILE NAME: JOHN

4 { INVALID FILE NAME
   FILE NAME IN USE

   NEW FILE NAME: MARY

   READY

5 { 10 INPUT B,S
   20 IF S>100 THEN 50
   30 LET C=0
   40 GO TO 60
   50 LET C=0.04*(S-100)
   60 LET P=B+C
   70 PRINT "TOTAL PAY IS" P
   80 END

6 { SAVE

   READY

7 { RUN

   ERROR 08
   UNMATCHED PARENTHESES LINE 60

   READY

8 { 60 LET P=B+C
   SAVE

   READY

9 { RUN

   ?
   82.50,476.13

   TOTAL PAY IS 458.67

   READY

10 { 50 LET C=0.04*(S-100)
   SAVE

   READY

11 { RUN

   ?
   82.50,476.13

   TOTAL PAY IS 97.545

   READY

12 { BYE

   OFF AT 10:51

```


7. The command RUN instructs the computer to execute the program. At this point, our system first scans the program for errors in syntax. Any error messages are printed followed by READY. No actual execution is attempted until all syntax errors are removed.
8. The error in line 60 is corrected by simply re-entering the entire line, which corrects the copy of the program in the computer's memory. Entering SAVE also corrects the program stored in the auxiliary storage unit. READY is again typed when the SAVE operation is completed.
9. Upon the next entry of RUN, no syntax errors are found, and execution begins. After the computer types a ? character, the programmer enters the values required by the INPUT statement. The computer then prints the answer followed by READY to indicate that the computer is awaiting further instructions.
10. The observant programmer would realize that the answer is unreasonable and search for another bug. The + in line 50 is changed to an * by re-entering the entire line. Entry of SAVE also corrects the copy on the auxiliary storage unit.
11. The next run of the program produces the correct answer.
12. When the programmer has completed his work at the terminal, he enters BYE and the computer transfers his status from an active to an inactive state.

Most time-sharing systems provide the programmer with far more features than were used in the example in Figure 1-19. Many editing features are available at the terminal to facilitate such operations as listing the entire program or segments thereof, renumbering the lines in the program, and inserting or extracting groups of lines from the program.

For the small or occasional computer user, the conversational-time-sharing mode of operation is far more desirable than batch processing. For this reason, the popularity of conversational time-sharing has increased rapidly and probably will increase even more in the future. Batch processing is attractive only to the user with a very large program that requires virtually the entire machine in order to run and to the user whose program runs so long that the wait at the terminal would approach the wait for turnaround in the batch-processing environment. The number of small users far exceeds the number of users in both of these situations.

2

ALGORITHMIC PROCESSES

Basically, an algorithm is a procedure for determining the solution to a problem, whether numerical or nonnumerical. In certain respects, an algorithm resembles a recipe in that it consists of a sequence of specific instructions that if followed obediently will lead to the desired end result. That algorithms are associated with computers should be obvious. The computer will follow instructions obediently and with phenomenal speed, making virtually no errors in the process.

While computers can be programmed to follow an algorithm, the responsibility of constructing the algorithm rests with the programmer. Unfortunately, there are no step-by-step procedures for developing an algorithm. The only available procedure is to look at some specific examples, which shall be our approach in this chapter.

2.1. A NONNUMERICAL ALGORITHM

As our first example, we shall consider a nonnumerical algorithm, specifically, that of the husband paying the monthly bills. When the time has come to pay the bills, he proceeds as follows:

1. Get the check book.
2. Get all unpaid bills.
3. Pay them.

The steps in this expression of the algorithm could be represented in more detail, which we shall do shortly. Furthermore, they do not take into account such situations as insufficient funds to pay all the bills. Thus, the algorithm in this form is not entirely satisfactory, and we shall make a number of improvements.

As illustrated by the example in Chapter 1, flowcharts are very useful in the representation of algorithms. Furthermore, these flowcharts tend to have certain

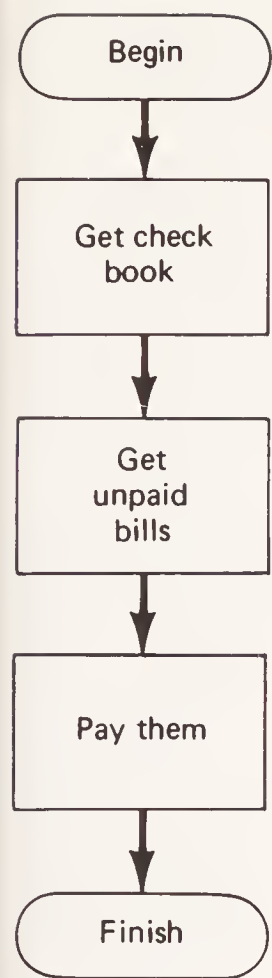


Figure 2-1 Flowchart of the bill-paying algorithm

distinguishing structures. The representation of our bill-paying algorithm in Figure 2-1 reveals a flowchart that is once-through in nature in that each step is executed only once without any “doubling-back,” that is, returning to previous steps.

Many flowcharts may be expressed in varying degrees of detail. For example, suppose we consider the block “Pay them” in the flowchart in Figure 2-1. If it is known beforehand that there will always be a certain number of bills each month, our flowchart might be constructed as illustrated in Figure 2-2. Here, a separate block is designated for each bill, in this case four. The flowchart still retains its once-through nature.

In the more usual case, the exact number of bills is not generally known beforehand, thus making it impossible to construct a flowchart such as given in Figure 2-2 that would be appropriate for all months. Instead, the flowchart in Figure 2-3 would be a more appropriate representation of the algorithm. This flowchart is no longer once-through in that execution of one of the blocks is repeated until a certain condition has been satisfied, namely, that all bills have been paid. The corresponding section of the flowchart is said to comprise a *loop*. Execution of all blocks within the loop is repeated until the condition permitting an exit is satisfied.

In many cases, the once-through type of flowchart can be more conveniently represented by one containing a loop. For example, the flowchart in Figure 2-3 could be used in place of the flowchart in Figure 2-2 for the case in which exactly four bills are received each month. If twenty bills instead of four were received each month, the once-through representation used in Figure 2-2 would

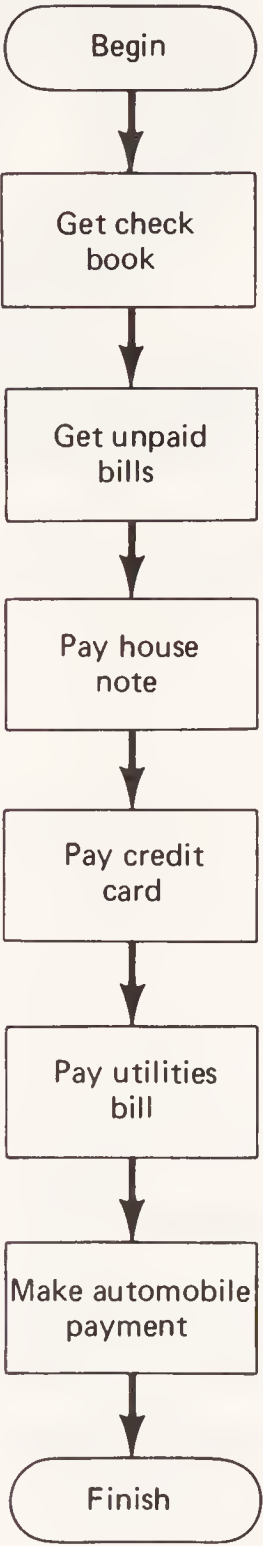


Figure 2-2 Expansion of flowchart in Figure 2-1

become somewhat awkward, whereas the loop representation in Figure 2-3 is independent of the number of bills.

A flowchart may contain any number of loops. If the husband has to sort through the mail, he might choose to follow the procedure in Figure 2-4, where he first sorts the mail to obtain all the bills and then proceeds to pay them.

The flowchart in Figure 2-4 also contains an example of two parallel paths, one of which is followed under certain conditions and the second followed otherwise. If a letter contains a bill, it is kept. If not, it is discarded.

A flowchart may also have more than one exit. If the checking account is depleted in the process of paying the bills, the depressed and frustrated husband might follow the alternate path illustrated in Figure 2-5.

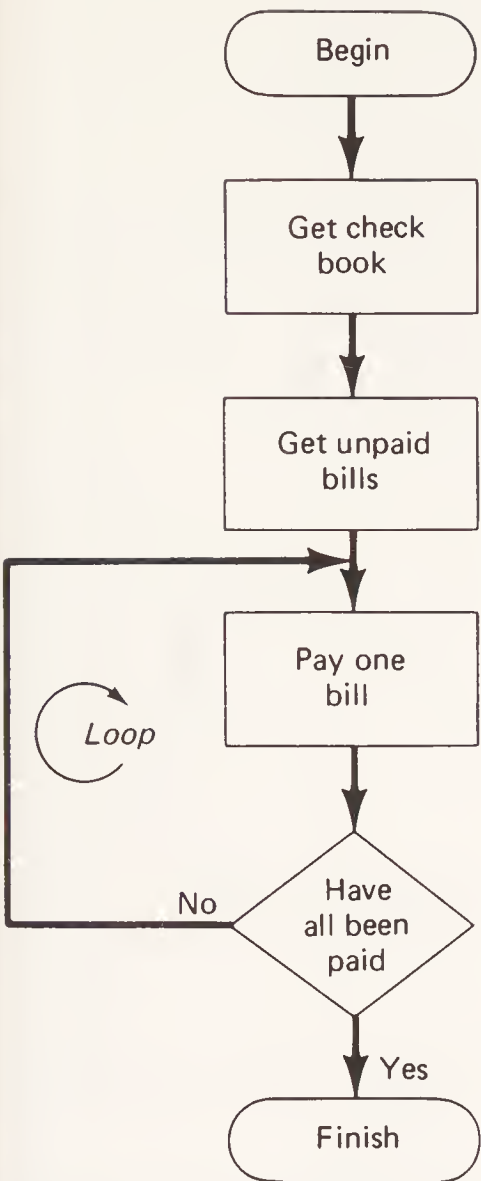


Figure 2-3 Flowchart containing a loop

2.2. AN EXAMPLE OF A NUMERICAL ALGORITHM

For any given problem, there may be several algorithms that yield the desired solution. Therefore, in one sense they are all correct, but one may require less effort to obtain the solution and thus should be preferred over the others.

To illustrate this point, consider constructing an algorithm to determine if the number 313791 is a prime number.[†] One algorithm for solving this problem is represented by the flowchart in Figure 2-6. We simply divide 313791 by consecutive integers, starting with 2 and continuing through 313790. If the result of any one of these is an integer, 313791 is not a prime number. If not, then 313791 is a prime number.

Clearly, this approach will give the desired answer, but considerable improvement can be made. First, why continue dividing by integers whose value exceeds

[†]Restricting our thinking to positive whole (integer) numbers, a prime number is one that is not evenly divisible by any other number except itself and the number 1. That is, 2, 3, 5, 7, 11, 13, etc., are prime, whereas 4 (divisible by 2), 6 (divisible by 2 or 3), etc., are not.

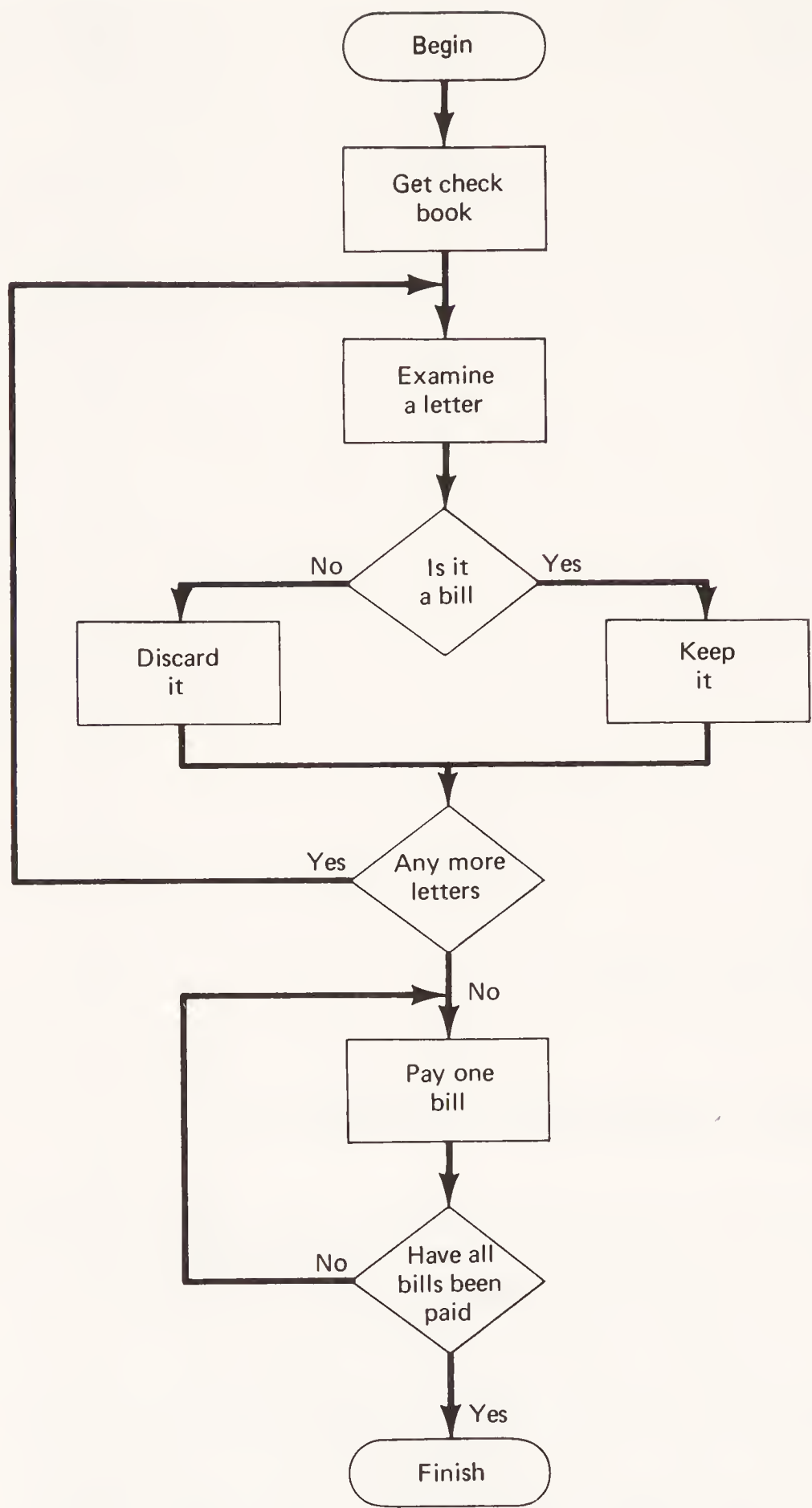


Figure 2-4 Flowchart with two loops

313791/2? For example, dividing 313791 by any integer number greater than 156895 ($= 313791/2$) but less than 313791 produces a nonintegral result. Thus, it is necessary to search over only the integers whose values are less than 313791/2, as is illustrated by the flowchart in Figure 2-7.

As compared with the algorithm in Figure 2-6, this approach reduces the number of calculations by almost a factor of 2.

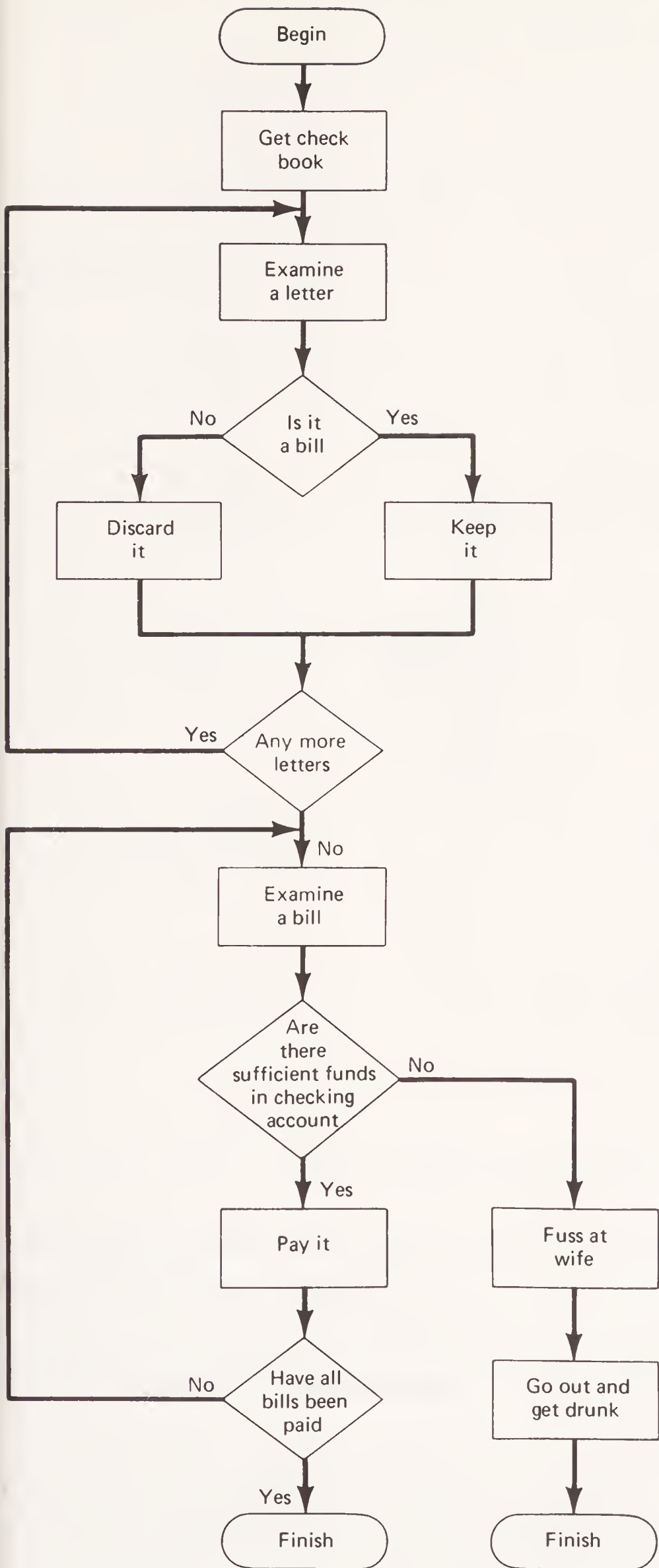


Figure 2-5 Flowchart with two terminals

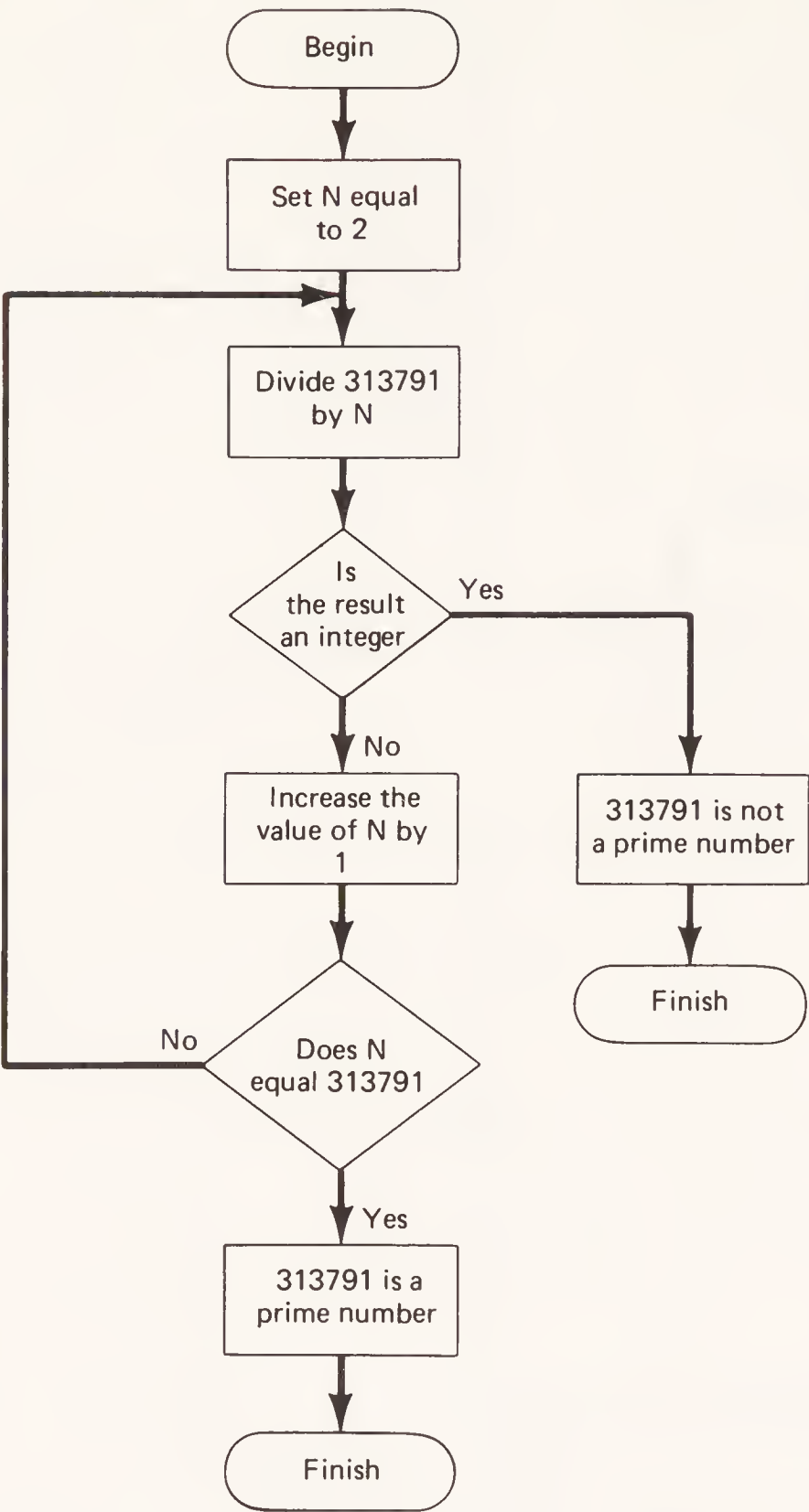


Figure 2-6 Algorithm to determine if 313791 is a prime number

Is further improvement possible? The answer is yes, in fact, even by another factor of about 2. After we divide 313791 by 2 and discover that the result is not an integer, why divide by even values of N ? That is, if 2 is not a divisor, neither are 4, 6, 8, etc. After considering 2 as a divisor, we then need only to consider 3, 5, 7, 9, etc. This logic is incorporated into the flowchart in Figure 2-8.

Are there more efficient algorithms for determining if 313791 is a prime number? How would you go about proving or disproving that a proposed algorithm is the most efficient possible? You could disprove this hypothesis by devising a more efficient algorithm, which should be easy in this case (Give it a try!). But if you are unsuccessful, does this prove that the algorithm is the

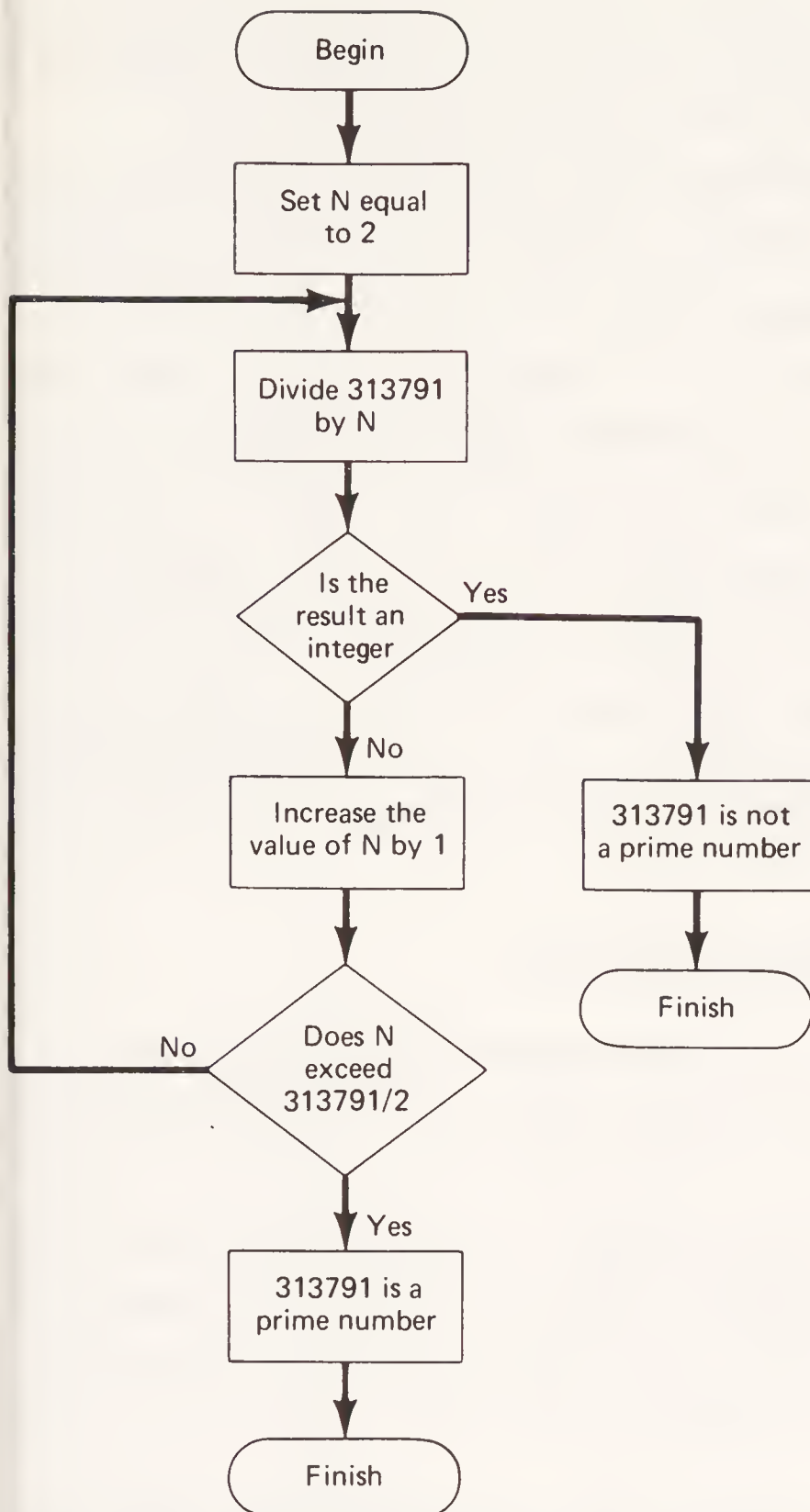


Figure 2-7 Improved algorithm to determine if 313791 is a prime number

most efficient possible? Unfortunately not, and in general such an hypothesis cannot be proven to be true.

In the prior discussions, we have applied the term *efficiency* only to the computations that must be undertaken. In a more realistic sense, the solution to a problem requires two distinct efforts:

1. The programmer must contribute some of his time in preparing the program for the computer.
2. The computer must devote some of its time to executing the program.

In the final analysis, the most efficient algorithm is the one that achieves the appropriate balance between programming effort and machine time. In this

regard, the programmer's time might be evaluated as so many dollars per hour and the machine's time at a different rate. The most desirable algorithm is the one that obtains the solution with the least total expenditure of money.

With regard to the three algorithms we have proposed for determining if the number 313791 is prime, the algorithm in Figure 2-6 is the simplest to program but the least efficient in terms of machine time. The algorithm in Figure 2-7 requires about half as much machine time and is only trivially more difficult to program. The algorithm in Figure 2-8 is twice again as efficient and requires only a small amount of additional programming effort. In today's market, the people-related resources tend to be the scarcer, which in many cases would tend to make one choose a less efficient algorithm that could be more easily implemented. Of course, extra programming effort is much easier to justify for a program that is to be run many times than for a program that is to be run only once. Students should not, however, use this statement to justify inefficient programming practices. In many cases, efficient methods are just as easy to program as inefficient ones; and even when this is not true, the experience in using efficient programming techniques certainly contributes to one's education.

2.3. PROGRAMMING THE ALGORITHM

Although we shall not go so far as to use a specific programming language, we want to examine certain characteristics common to most programming languages. Specifically, we shall choose the flowchart given in Figure 2-7 as the basis for our discussions.

In Chapter 1, we discussed the two phases, compilation and execution, that occur between the time a program is submitted and the time answers are received. One very important aspect of the compilation phase is the allocation of storage locations for use by our program. Basically, the compiler allocates two types of storage locations:

1. Storage locations for constants, that is, quantities whose values never change. The compiler also stores the appropriate constant value into these storage locations.
2. Storage locations for variables, that is, quantities whose values can change. We shall assume that the compiler does *not* store anything in these storage locations. (Actually, some compilers initially assign the value zero to all storage locations reserved for variables. Others provide special statements to assign the appropriate initial values to these storage locations.)

Interpreters function in a similar manner except that they need not reserve storage locations for constants.

To illustrate these points, suppose we use a typical statement of the form

$$N \leftarrow 2.706$$

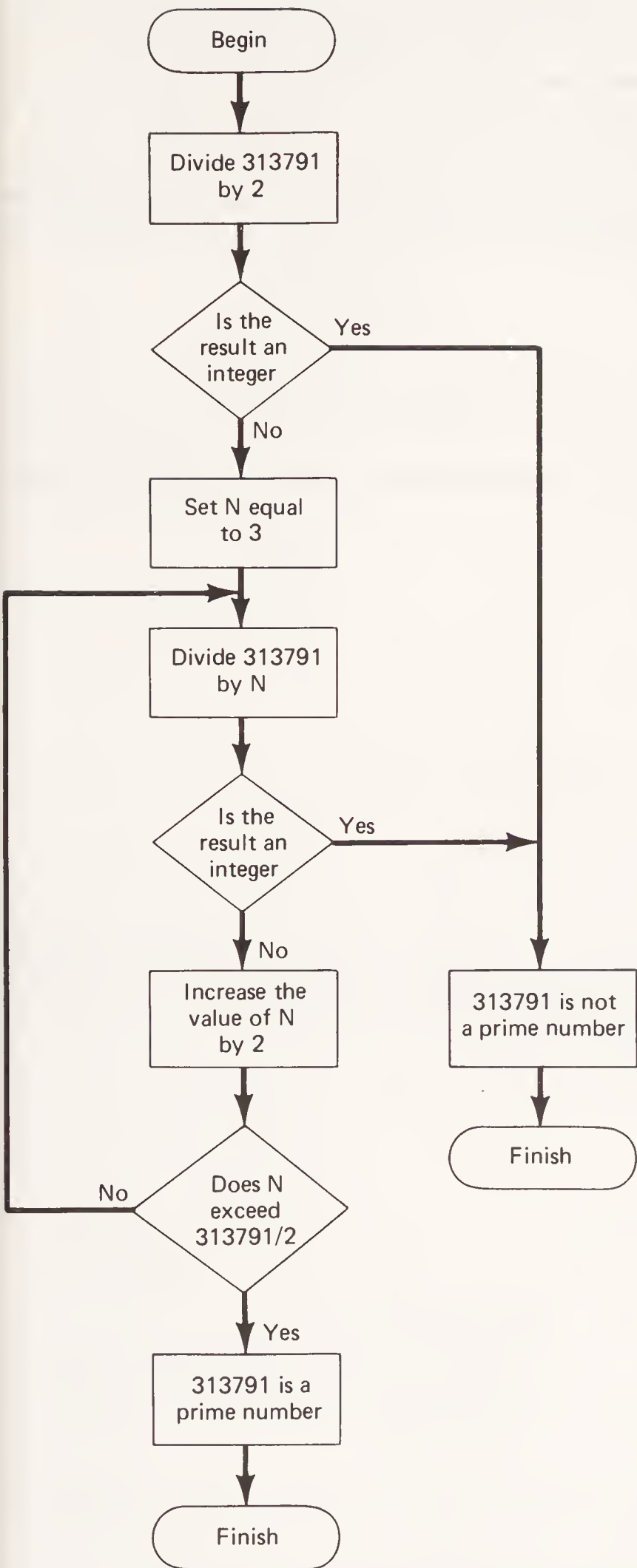


Figure 2-8 More efficient algorithm to determine if 313791 is a prime number

48 which instructs the computer to “assign the value 2.706 to N.” The compiler will set aside two storage locations, one for the constant 2.706 and another for the variable N. The compiler also stores the value 2.706 in the storage location reserved for the constant. For the variable N, the compiler *only* reserves a storage location; that is, it does not store anything in that storage location. Therefore, when the execution phase is entered, the variable N is said to be “undefined”; that is, no value has been stored in its storage location. In executing the statement

$$N \leftarrow 2.706$$

the computer inserts the value 2.706 into the storage location for N, thus “defining” N.

In our flowcharts, we shall permit constants to be written in any one of the following ways:

- 1. Without a decimal point:

$$1070$$

- 2. With a decimal point:

$$8.726$$

- 3. In exponential form:

$$1.276 \times 10^{-6}$$

As variable names, we shall permit any string of the twenty-six alphabetic characters and the ten digits as long as (1) the first character is alphabetic and (2) there are no embedded blank spaces. Acceptable names are

JOEKOLLAGE
A
X123456
X1X2

Unacceptable names include

JOE KOLLAGE	(embedded blank)
1X	(first character not alphabetic)
H - X	(character — not permitted as part of variable name)

As we shall see later, H - X will designate subtraction of the value of variable X from the value of variable H.

The statement first introduced in most programming courses is the arithmetic assignment statement whose form is

$$variable \leftarrow arithmetic\ expression$$

The *arithmetic expression* is essentially any combination of variables, constants, and arithmetic operations permissible within the programming language. In such expressions, the following five operations will be permitted:

Symbol	Operation	Programming example	Algebraic equivalent
+	Addition	$A + B$	$a + b$
-	Subtraction	$A - B$	$a - b$
$\times$	Multiplication	$A \times B$	$a \cdot b$
/	Division	A / B	a / b
$\uparrow$	Exponentiation	$A \uparrow B$	a^b

The following expressions are valid:

$X + B$
 $A \times C$
 $A + B/4.7$
 G
 7.9

The last two examples indicate that an expression may be composed of a single variable or a single constant. The following expressions are not valid:

$X + B +$ (ends with an operator)
 $Q \times / B$ (two operators in succession)

We shall consider in a later section specific rules for hierarchy of arithmetic operations along with the use of parentheses.

Expressions may also contain functions, which are special routines for computing such quantities as square roots, absolute values, and trigonometric functions. In our examples, we will recognize the following functions (names are actually those from Basic):

Function	Explanation
SQR(X)	Square root, $\sqrt{x}$
ABS(X)	Absolute value, $ x $
SIN(X)	Trigonometric sine, $\sin x$, x in radians
COS(X)	Trigonometric cosine, $\cos x$, x in radians
ATN(X)	Trigonometric arctangent, $\tan^{-1} x$, result in radians
LOG(X)	Natural (base e) logarithm, $\ln x$
CLG(X)	Common (base 10) logarithm, $\log_{10} x$
EXP(X)	Exponential function, e^x

We shall introduce others in subsequent sections.

The quantity in parentheses following the name of the function is called the

50 *argument*. In addition to being a simple variable or constant, the argument of a function may be any valid expression. In these cases, the numerical value of the expression is calculated and then the functional value computed. The following examples are valid:

```
SQR (4.2) + A
B/ABS (A + B)
SQR (ABS(G/2) + 4.7)
SQR (SQR (D × Y))
```

The last example indicates that a function may appear in the expression used as the argument of the same function.

In executing the arithmetic assignment statement, the computer first evaluates the expression to the right of the assignment operator ($\leftarrow$) and then stores the numerical result in the storage location reserved for the variable on the left of the assignment operator. The following candidates do not conform:

```
7.2 ← B + C      (constant on the left of the assignment
                  operator)
A + 1.4 ← B + C   (expression on the left of the assignment
                  operator)
```

It is possible, and, in fact, quite common, for the variable on the left of the assignment operator to appear in the expression on the right. The evaluation of the expression is performed with the current values for all variables appearing therein, followed by the storing of the result in the storage location for the variable appearing on the left of the assignment operator. As an example, consider the following statements:

```
A ← 1.2
A ← A + 5
```

The resulting value of A is 6.2.

For the flowchart in Figure 2-7, statements of this type are used to increment *N*. Incorporating these statements gives the flowchart in Figure 2-9.

Many programming languages (notably Fortran, Basic, PL/I, and Cobol) use an equals sign where we have used the assignment operator. That is, our statement

```
A ← A + 5
```

would be programmed in these languages as follows:

```
A = A + 5          Fortran; PL/I
LET A = A + 5      Basic
COMPUTE A = A + 5  Cobol
```

In each case, the equals sign is used to represent the assignment operation, not algebraic equality. That is, the statement

```
A = A + 5
```

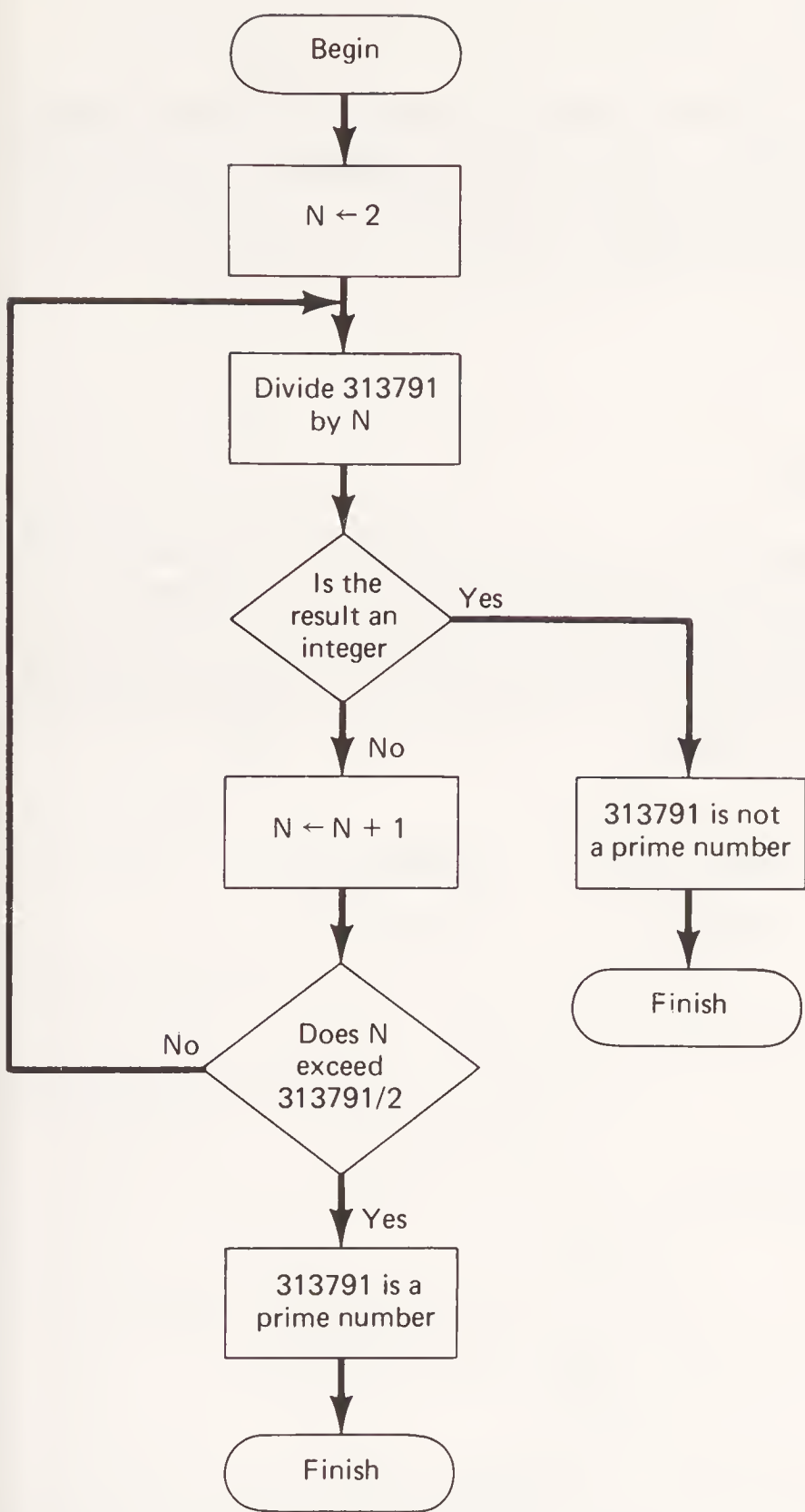


Figure 2-9 Use of arithmetic assignment statements in the prime-number algorithm

in algebra would be meaningless. Subtracting A from both sides gives

$$0 = 5$$

which is not true. On the other hand, the statement

$$A \leftarrow A + 5$$

means “take the current value of variable A, increase its value by 5, and store the result in the storage location reserved for variable A.”

Thus, the assignment operator in arithmetic assignment statements really should be read “is replaced by” and not “is equal to” even in those programming languages that use the equals sign as the assignment operator.

52 2.4. DECISION BLOCKS

The flowchart in Figure 2-9 contains two decision blocks, one for ascertaining if the result of a division is an integer and another for ascertaining if N exceeds $313791/2$. In order to make such decisions, the question must frequently be rephrased, since the computer is basically capable of answering only the following three questions:

- 1. Is a number positive?
- 2. Is a number negative?
- 3. Is a number equal to zero?

Although these may at first appear to be restrictive, quite complex decisions can be expressed as sequences of these three coupled with arithmetic operations.

Most implementations of the decision block make use of a relational operator to construct a logical expression that will either be true or false. The general form of the logical expression is as follows:

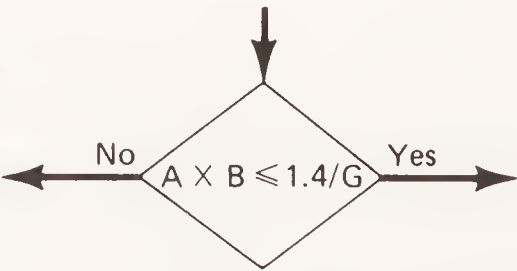
$$\left\{ \begin{array}{l} \textit{arithmetic} \\ \textit{expression} \end{array} \right\} \quad \left\{ \begin{array}{l} \textit{relational} \\ \textit{operator} \end{array} \right\} \quad \left\{ \begin{array}{l} \textit{arithmetic} \\ \textit{expression} \end{array} \right\}$$

we shall permit the following six relational operators:

Operator	Interpretation
<	Is less than
>	Is greater than
≤	Is less than or is equal to
≥	Is greater than or is equal to
=	Is equal to
≠	Is not equal to

When these operators are used to construct decision blocks, only two paths may emanate from the block, one of which is labeled true, or “yes,” and the other labeled false, or “no.” Three examples of decision blocks in this category are given in Figure 2-10.

To illustrate how the computer executes decision blocks using the three tests given in the first paragraph in this section, consider the block



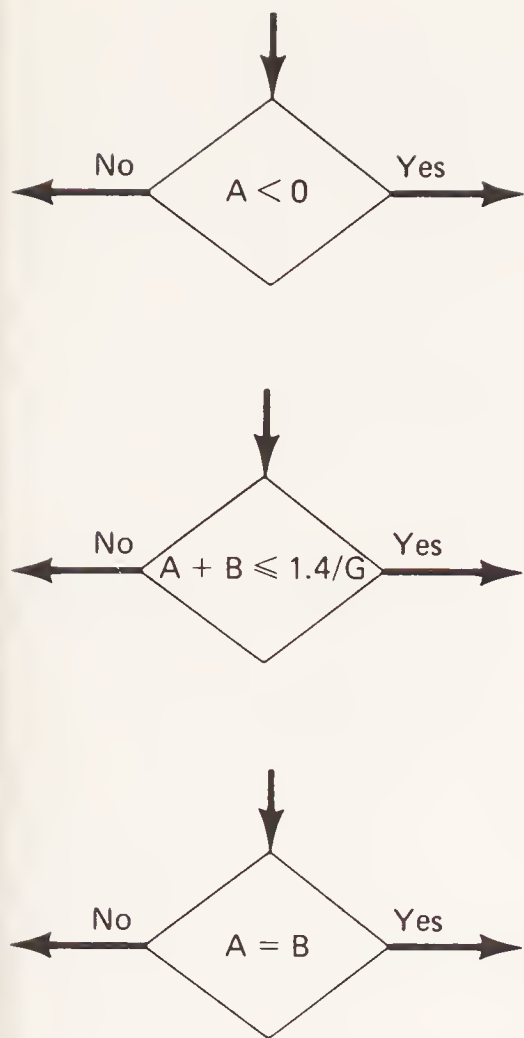


Figure 2-10 Examples of decision blocks

The logical expression is rearranged to

$$A \times B - 1.4/G \leq 0$$

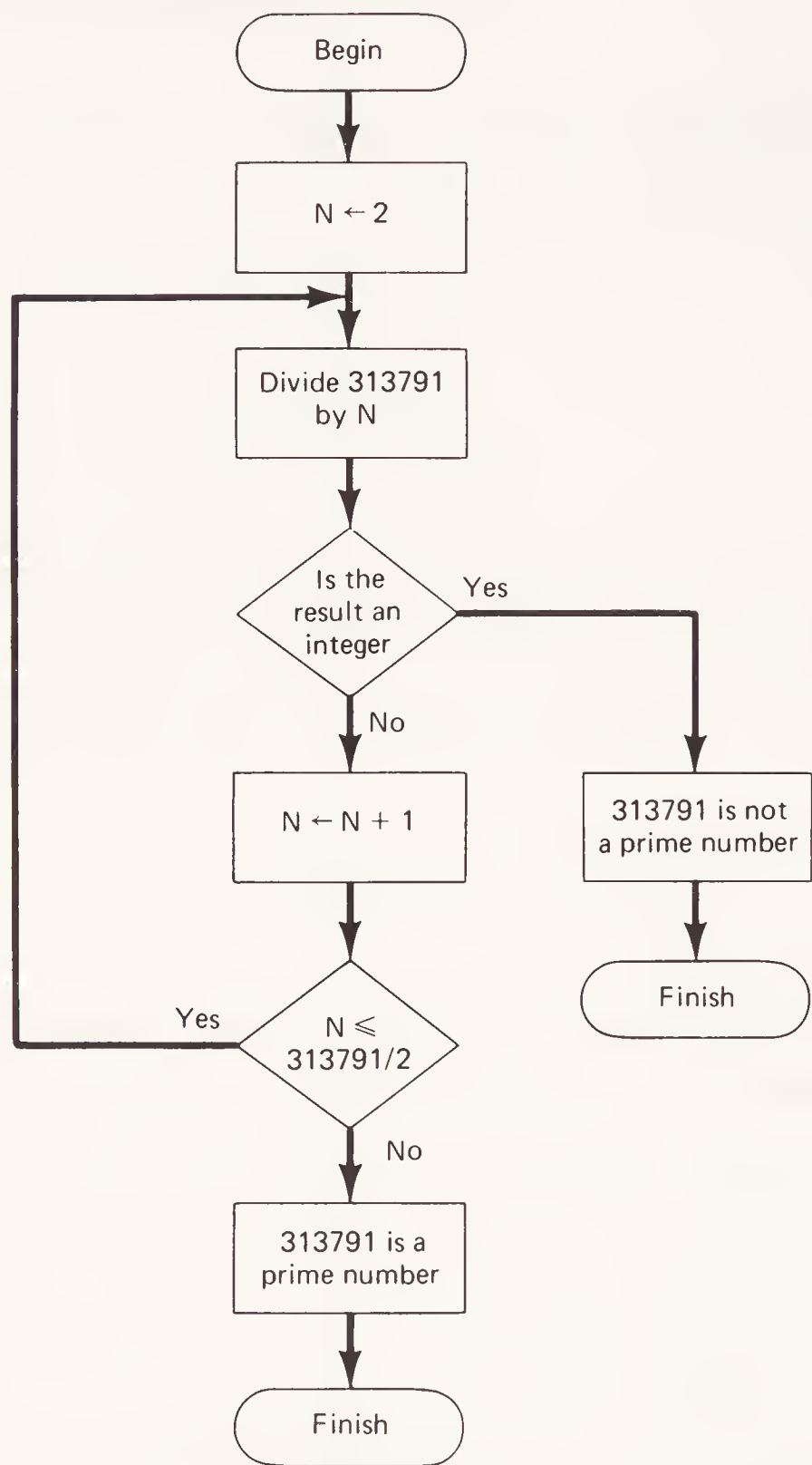
The arithmetic expression

$$A \times B - 1.4/G$$

is evaluated. If the result is negative or zero, the “yes” branch is taken. If the result is positive, the “no” branch is taken.

Figure 2-11a presents a version of the prime-number algorithm with a decision block used at the end of the loop. Although this flowchart would work perfectly well, let us again consider the question of efficiency for a moment. Execution of the decision block in Figure 2-11a requires the computation of $313791/2$. If programmed and executed in exactly the manner shown, this computation would be repeated on each pass through the loop, even though the result never changes. If $313791/2$ were computed and stored prior to entering the loop as is illustrated in Figure 2-11b, all but one of the division computations are eliminated.

In other words, all computations whose result does not change during execution of the loop should be placed ahead of the loop. Some compilers that attempt to optimize the code will in some situations do this for you, but many do not. This might seem to be a trivial savings in computer time in this

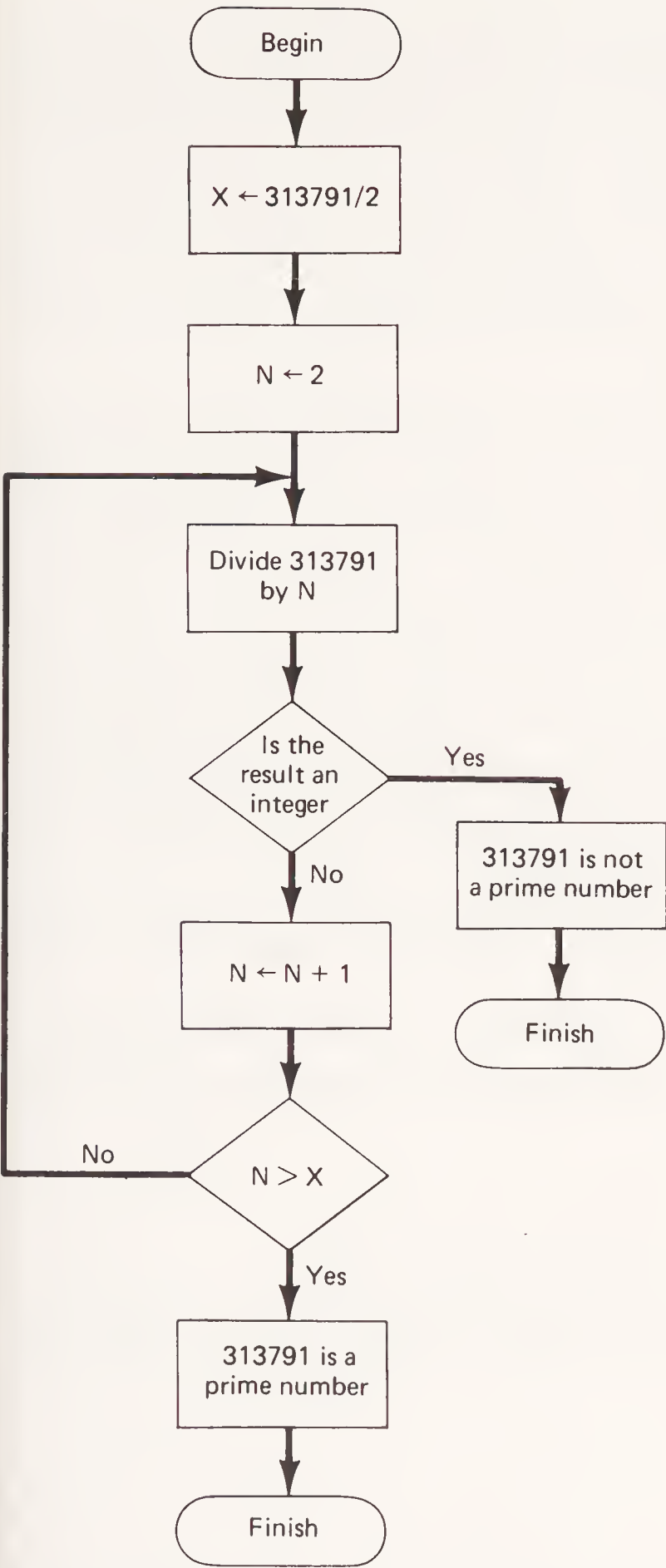


a. Prime-number algorithm with logical-decision block at end of loop
Figure 2-11 Use of a logical-decision block

particular case, since the typical computer can perform several thousand divisions in one second. It is good programming practice, however, and should be recognized by the student early in his career.

2.5. TRUNCATION AND ROUNDING

In one of the decision blocks in our prime-number algorithm, the computer must ascertain if the result of a division is an integer, that is, has no fractional part. Again, we must construct a procedure which enables the computer to obtain the answer by using a logical expression in a decision block. This process is accomplished with the aid of a function that deletes the fractional part.



b. Improved version
Figure 2-11 (Continued)

Consider the number 2.8205, which contains a fractional part, 0.8205. This number can be converted to an integer in one of two ways. First, the fractional part could be simply deleted or chopped off to give the integer 2, a process known as *truncation*. Alternatively, we might note that the integer whose value is closest to 2.8205 is 3, a process known as *rounding*. As we shall see below, both of these processes have their places.

Suppose a boy enters a department store to buy some baseballs. He takes

exactly one dollar and discovers that the baseballs he wants cost 35 cents each. How many baseballs does he get? Dividing \$1.00 by \$0.35 and disregarding sales tax, we discover that he gets $2 \frac{30}{35}$, or 2.857, baseballs. But the salesperson is not going to sell only part of a baseball, and even if he did, a baseball with a chunk taken out of it is worthless. Even as much as the salesperson might like to, today's business practices are such that our boy is not likely to get three baseballs for anything less than \$1.05. Therefore, he gets two baseballs and 30 cents in change. In this example, truncation, not rounding, is the appropriate process for deleting the fractional part.

Occasionally my wife asks me to measure the width of a piece of furniture such as a table. Am I going to tell her it is exactly $27 \frac{3}{16}$ inches wide? She doesn't really care about the fraction, so I would give her the nearest number of inches, which in this case would be 27. If the table measured $27 \frac{9}{16}$ inches, I would tell her 28. This is the process of rounding.

In some cases, neither of these processes is appropriate. Suppose a man wants to install a water pipe from point *A* to point *B*, which measure 42 feet apart. Upon going to the hardware store, he finds that the pipe he wants to install can be purchased only in 20-foot lengths. How many must he buy? Dividing 42 by 20 indicates that he needs 2.1 units. Either truncating or rounding gives 2, but this gives him 40 feet instead of the needed 42. He must buy 3 units, even though he has no need for most of one unit. This is a process sometimes referred to as "rounding upward," that is, finding the nearest integer whose value *exceeds* the fractional number.

In our flowchart language, we shall obtain these operations with the aid of a function which we shall call INT. In our usage, this function always truncates or deletes the fractional part. That is,

$$\text{INT}(5.727) = 5$$

We shall see later that this function can also be used to round.

To illustrate the use of this function, let's return to our prime-number algorithm. In determining if 313751 is exactly divisible by *N*, we might first compute *F* as follows:

$$F = 313791/N$$

Now we might ask the question

$$\text{Does } \text{INT}(F) \text{ equal } F?$$

If so, then *F* does not have a fractional part and 313791 is not a prime number. If not, then *F* has a fractional part and 313791 is still a candidate for a prime number. The final flowchart is illustrated in Figure 2-12.

Another manner of phrasing the question in the preceding paragraph is as follows:

$$\text{Does } \text{INT}(313791/N) \times N \text{ equal } 313791?$$

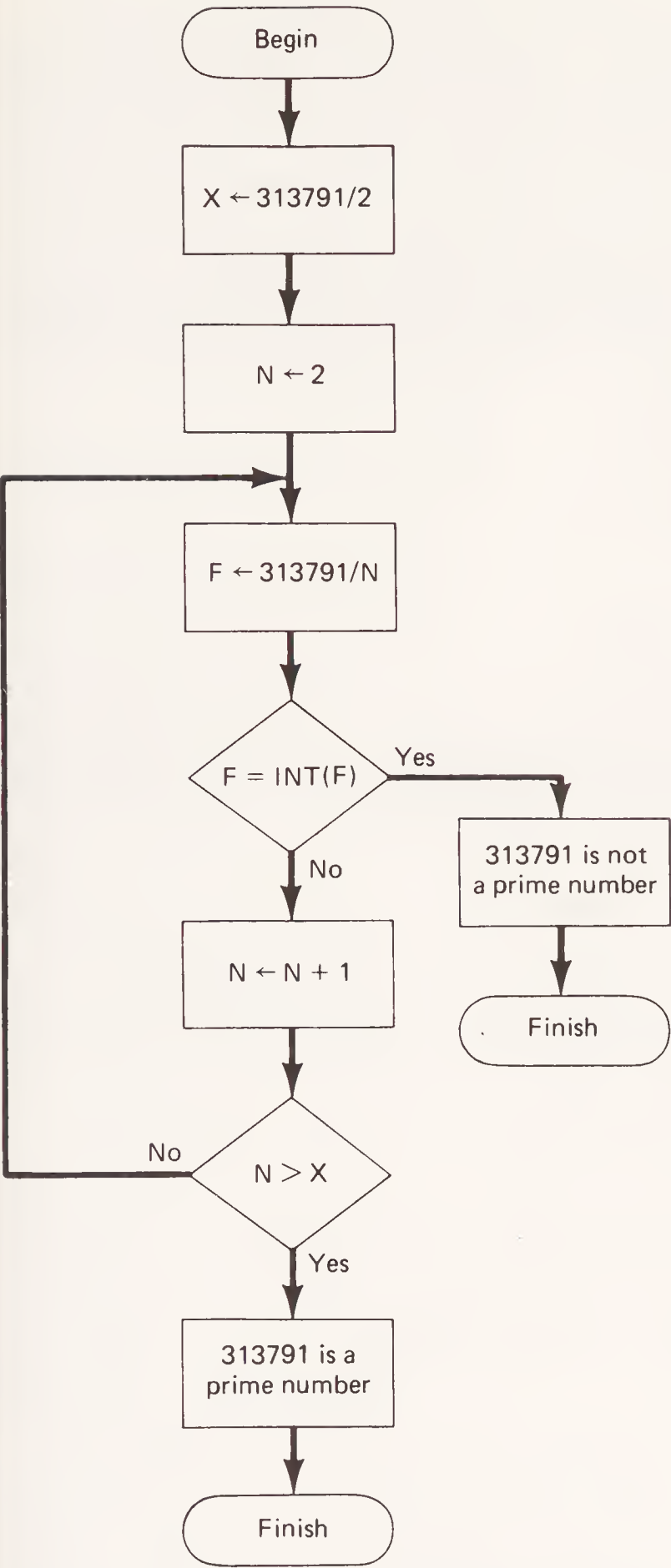


Figure 2-12 Use of the INT function in the prime-number algorithm

In most programming languages, this form of the expression would permit the decision block to be implemented in a single statement.

To obtain rounding with the INT function as it has been presented here, the value of the argument must be either increased or decreased by 0.5, depending upon whether the argument is positive or negative. For positive values of X , the appropriate expression is

$$\text{INT}(X + 0.5)$$

If X equals 1.4, the result is 1. If X equals 1.6, the result is 2. For negative values of X , the expression should be

$$\text{INT}(X - 0.5)$$

If X equals -1.4 , the result is -1 . If X equals -1.6 , the result is -2 .

Although a decision block could readily select the proper alternative, a single statement can be obtained by using the function $\text{SGN}(X)$ defined as follows:

$$\text{SGN}(X) = \begin{matrix} 1 & \text{for } X > 0 \\ 0 & \text{for } X = 0 \\ -1 & \text{for } X < 0 \end{matrix}$$

The statement

$$\text{INT}(X + 0.5 \times \text{SGN}(X))$$

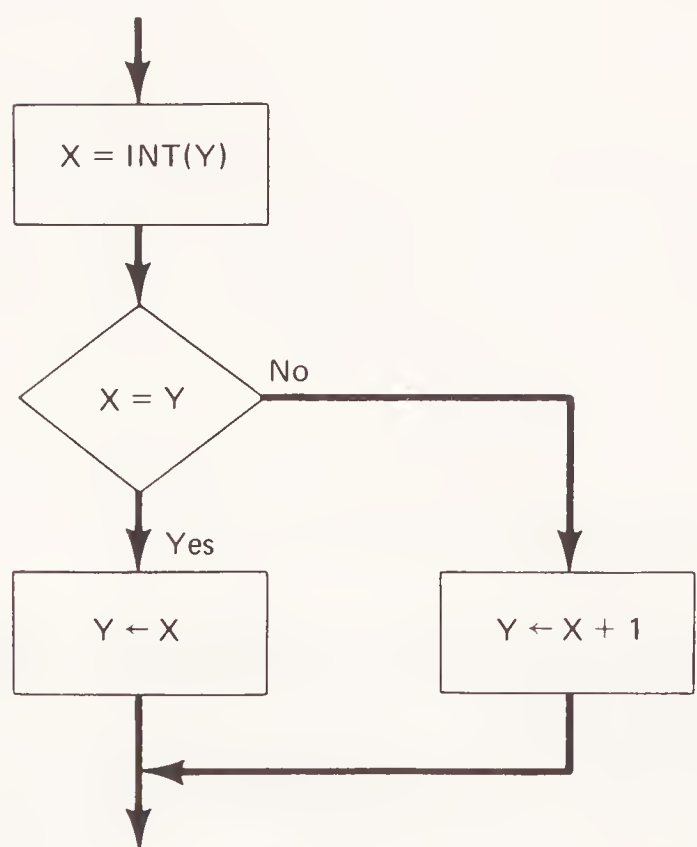
rounds for X positive or negative.

This approach can also be extended to rounding to a certain number of decimal places. For example, rounding to the nearest tenth is obtained by the statement

$$\text{INT}(10 \times X + 0.5 \times \text{SGN}(X))/10$$

A similar statement could be developed for rounding to N decimal places.

The rounding-upward process generally requires a decision block such as the following (for rounding Y upward):



Since Y might have originally been an integer, it is not permissible always to add 1.0 to Y and then truncate.

As a final note, a few comments are appropriate regarding implementations of the INT function or its equivalent in various programming languages. First, Fortran and some versions of Basic have the function INT almost exactly as described here. In many, however, the expression $\text{INT}(X)$ “determines the largest integer value that does not exceed the value of X .” For positive X , this is equivalent to truncation. For negative X , however, the result is different. That is, truncating -4.7 gives -4 . However, the largest integer whose value does not exceed -4.7 is -5 . In this case, the statement

$$\text{INT}(X + 0.5)$$

rounds for X positive or negative.

2.6. INPUT/OUTPUT

Communication with the user is an important aspect of any program. In most cases, this must be a two-way communication: the user must supply parameters and other input data to the program, and the program must relay the results back to the user. In many business applications, these aspects overshadow all others.

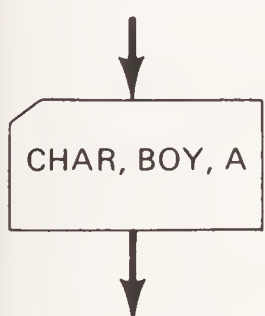
Data and parameters can be provided in any one of three ways or combinations thereof:

1. Special initialization statements are available on some systems that assign values to variables as part of the compilation process.
2. Parameters may be initialized by arithmetic assignment statements.
3. Data may be read from peripheral devices.

The first two approaches are attractive for parameters whose values are not expected to change. The assignments are done at virtually the speed of light. The first approach is somewhat preferred in that some executed instructions are eliminated from the compiled program, which causes it to require less storage and execute slightly faster.

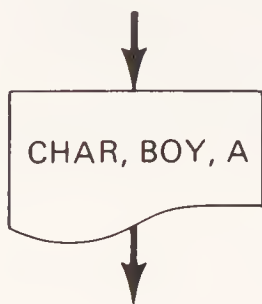
For parameters that change from run to run, reading the values from peripheral units is usually preferred. In this way, the different cases can be run without making any modifications to the program itself.

In our flowchart examples, we shall always use a card-shaped box to indicate input. The names of the variables to be read will be written inside the box and separated by commas, as illustrated by the following example:



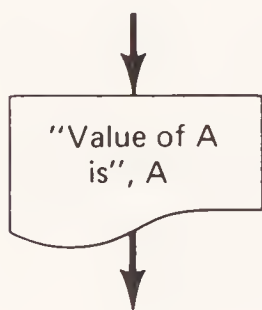
60 The use of a READ statement to enter the number 313791 in our prime-number algorithm is illustrated in Figure 2-13.

Although computers can accommodate several different output devices including teletype, cathode-ray tube (CRT), line printer, card punch, and magnetic tape, we shall always use the following box to indicate output:



Technically, this box denotes printed output as produced by either a teletype or a line printer. The variables whose values are to be printed are written inside the box and separated by commas. The output consists of *only* the value of the variable; that is, the name of the variable is not printed.

In order to print some explanatory information along with the values of the variables, we shall enclose the information in quotations and write it inside the box. That is, the box



prints "Value of A is" followed by the numerical value of A. Any sequence of symbols may be enclosed in the quotation marks and, other than for their appearance in the output, have no effect on the program.

Two examples of output boxes are given in Figure 2-13.

2.7. HIERARCHY OF OPERATIONS
IN ARITHMETIC EXPRESSIONS

One characteristic of languages for computers is that they must be completely unambiguous. In a previous section, we stated that arithmetic expressions were evaluated according to the rules of algebra. For example, the expression

$$A + B \times C$$

could be interpreted in either of two ways:

- 1. Multiply B by C and add the result to A, i.e., $a + (b \cdot c)$.
- 2. Add A to B and multiply the result by C, i.e., $(a + b) \cdot c$.

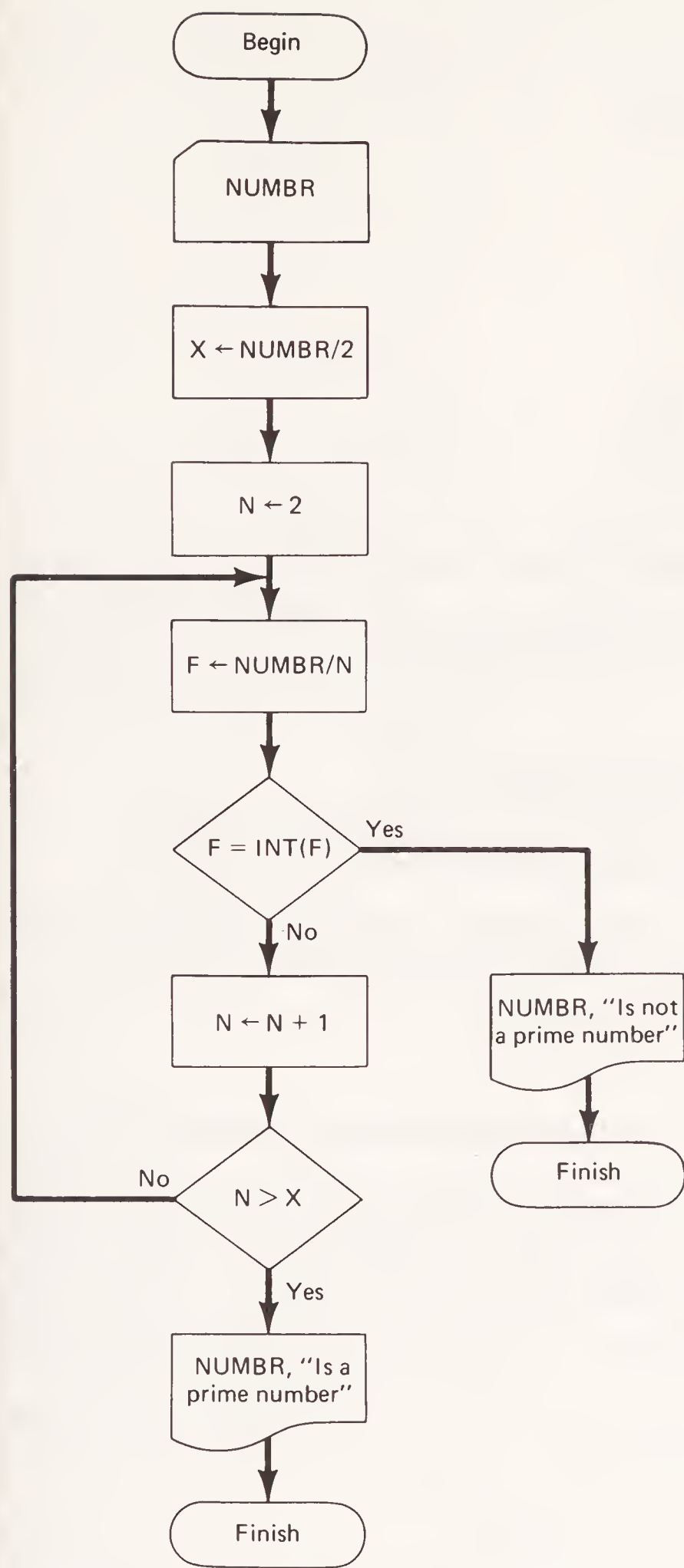


Figure 2-13 Use of input/output statements

The algebraic convention would be that the former interpretation is the proper one and is, in fact, the interpretation the computer would use. However, the expression

$A/B/C$

62 could be interpreted as follows:

$$\frac{a}{b/c} \qquad \text{or} \qquad \frac{a/b}{c}$$

Similarly, the expression

$$A/B \times C$$

could be interpreted as either

$$\frac{a}{b \cdot c} \qquad \text{or} \qquad \frac{a}{b} \cdot c$$

In these cases, the general rules of algebra are not very explicit.

While the rules of algebra are not very clear in situations like this, the rules by which the computer evaluates expressions are quite definitive. The first rule followed is that of hierarchy of operations, which is

Priority	Operation
1	Exponentiation, i.e., $\uparrow$
2	Multiplication and division, i.e., $\times$ and $/$
3	Addition and subtraction, i.e., $+$ and $-$

That is, all exponentiations are performed first, all multiplications and divisions second, and all additions and subtractions last. By these rules, the expression

$$A + B \times C$$

is interpreted as “multiply B by C and add the result to A,” or $a + (b \cdot c)$.

This hierarchy is not very useful, however, in expressions such as $A/B/C$. When more than one operation on the same level is to be performed, they are performed from *left to right*. For the above expression, this gives the interpretation “A divided by B and the result divided by C,” or $(a/b)/c$. Similarly, the expression $A/B \times C$ is interpreted as “A divided by B and the result multiplied by C,” or $(a/b) \cdot c$.

These rules can be overridden by the use of parentheses. To compute $(a - b) \cdot c$, the appropriate expression is

$$(A - B) \times C$$

To compute a/bc , the expression is

$$A/(B \times C)$$

Operations occurring within parentheses are performed first.

Table 2-1 gives several examples of valid expressions.

Table 2-1 Examples of Valid Expressions

$x \cdot (-y)$	$X \times (-Y) \text{ or } - X \times Y$
$-(x + y)$	$-(X + Y) \text{ or } - X - Y$
x^{i+1}	$X \uparrow (I + 1)$
$xy^{+1} \cdot z$	$X \uparrow (Y + 1) \times Z$
$\frac{x \cdot y}{z \cdot s}$	$X \times Y / (Z \times S) \text{ or } X \times Y / Z / S$ $\text{or } X / Z \times Y / S$
$\left(\frac{x + y}{z}\right)^{3.14}$	$((X + Y) / Z) \uparrow 3.14$
$x [r + y (r + z)]$	$X \times (R + Y \times (R + Z))$
$1 + \frac{x}{16.2 + z}$	$X / (1 + Y / (16.2 + Z))$

Before we leave the subject of expressions, it should be noted that the minus sign can be used in the following three ways:

Use	Example
To indicate subtraction	$A - B$
To indicate negation (usually referred to as a <i>unary</i> minus)	$-A$
To indicate a negative constant	-5.27

In the last case, the minus sign is actually part of the constant.

2.8. COMPOUND CONDITIONALS

Previous sections have presented decision blocks involving single logical decisions. While a combination of these can be used to accomplish virtually any decision, two or more can be combined into a single decision block via a compound conditional.

Suppose a function $f(x)$ is represented as in Figure 2-14. Mathematically, $f(x)$ is expressed as follows:

$$f(x) = \begin{cases} x^2 & -1 < x < 1 \\ 1 & x \leq -1, x \geq 1 \end{cases}$$

Suppose we construct an algorithm to read a value of x and determine the value of $f(x)$. If we use simple conditionals, the flowchart is as shown in Figure 2-15a.

Since the branches from two of the simple decision blocks lead to the same point, these can be combined into a compound conditional as shown in Figure 2-15b. The compound conditionals are constructed by the use of one of the logical operators AND, OR, or NOT. They are used to combine one or more relational expressions such as $X \geq 1$ into a combined conditional.

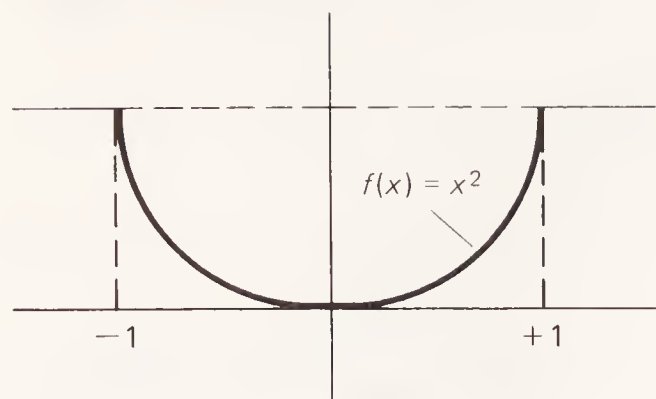
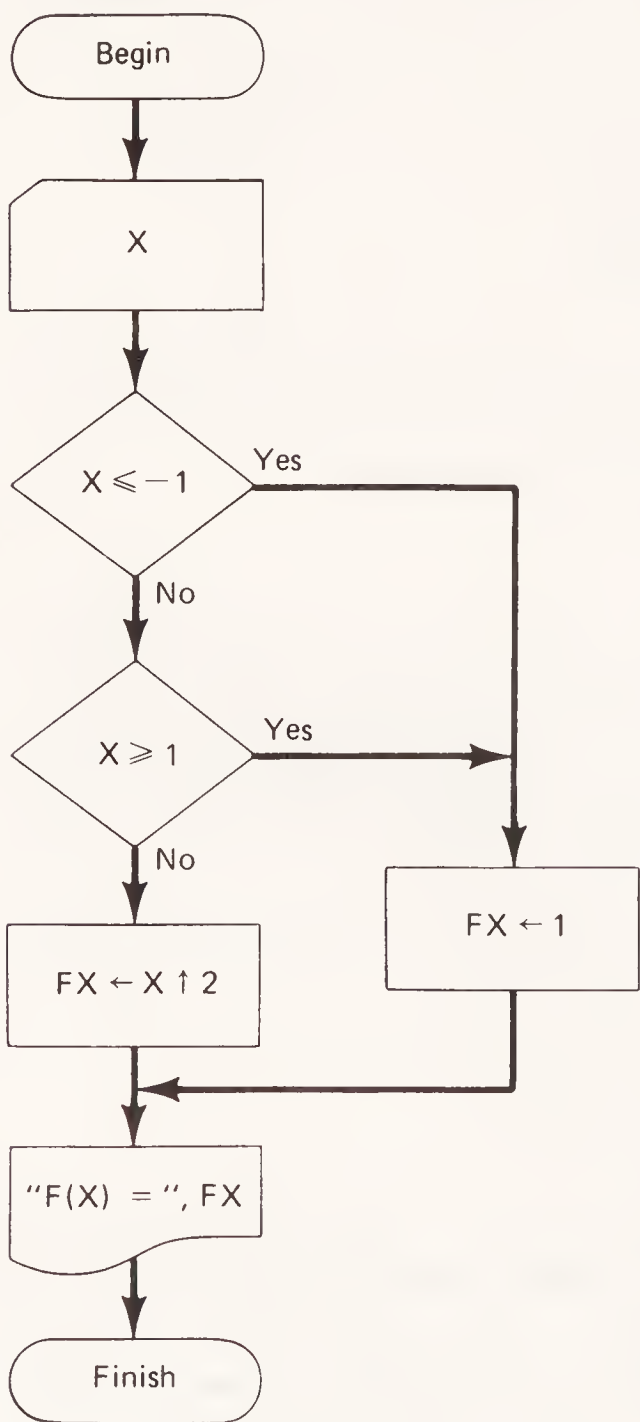


Figure 2-14 Function $f(x)$

Suppose A and B are two relational expressions and thus can only assume the values TRUE or FALSE. The compound conditional

A AND B



a. Using simple conditionals

Figure 2-15 Algorithm for computing $f(x)$

is TRUE if and only if both A and B are TRUE. If either A or B is FALSE, then the compound conditional is FALSE.

The compound conditional

A OR B

is TRUE if either A, B, or both A and B are TRUE. If both A and B are FALSE, then the compound conditional is FALSE.

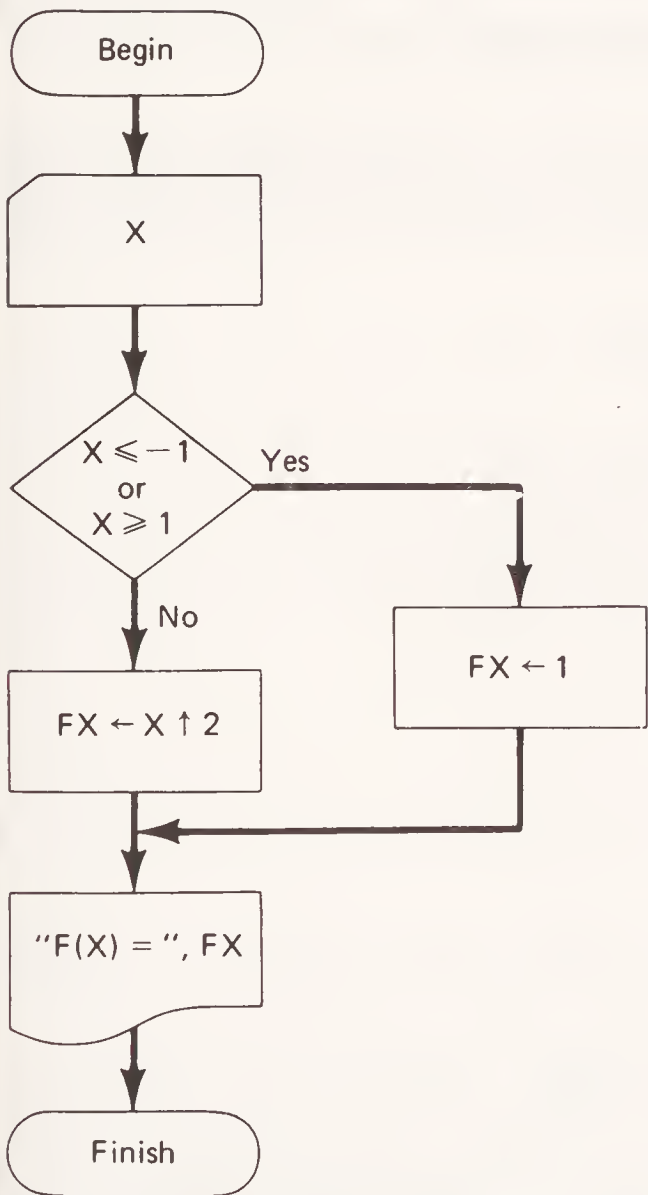
If A is TRUE, the conditional

NOT A

is FALSE; if A is FALSE, the conditional is TRUE.

The order of execution for the various operations appearing in a compound conditional statement can become confusing. Basically, the logical operators AND, OR, and NOT connect relational expressions, which are in turn composed of a pair of arithmetic expressions. Insisting that the relational expressions included in compound conditionals be enclosed in parentheses usually leads to greater clarity, for example,

NOT(A + B ≤ C) OR (B = A) AND (C ≤ 0)



b. Using compound conditionals
Figure 2-15 (Continued)

66 For statements such as this, the order of precedence for the logical operators is as follows:

Order of precedence	Logical operator
1	NOT
2	AND
3	OR

Therefore, the above compound conditional would be interpreted as follows:

$$(NOT(A + B \leq C)) OR ((B = A) AND (C \leq 0))$$

As in arithmetic expressions, parentheses can be freely used in compound conditionals.

EXERCISES

Note: Although only the preparation of flowcharts was the subject of this chapter, suggested input data is included for many exercises under the assumption that many students will be learning a programming language along with the study of flowcharts, and therefore will be programming some of these exercises.

2-1. Prepare the flowchart for a program to read a measurement in feet and print this measurement in centimeters, meters, inches, feet, and yards. The conversion factors are

$$\begin{aligned} 1 \text{ foot} &= 12 \text{ inches} \\ 1 \text{ yard} &= 3 \text{ feet} \\ 1 \text{ inch} &= 2.54 \text{ centimeters} \\ 1 \text{ meter} &= 100 \text{ centimeters} \end{aligned}$$

The output should be similar to the following (for 3 feet):

0.9144 METERS
91.4 CENTIMETERS
36 INCHES
3 FEET
1 YARDS

Run the program for 1.7 feet.

2-2. Prepare the flowchart for a program to read a volumetric measurement in cubic feet and convert to gallons (1 cu ft = 7.48 gal), liters (1 cu ft = 28.316 liters), cubic yards (27 cu ft = 1 cu yd), bushels (1 cu ft = 0.8036 bushels), and U.S. liquid barrels (1 bl = 31.5 gal). The output should be analogous to that in the previous exercise. Run the program for 10 cu ft.

2-3. A man borrows P dollars for n years at an interest rate of i . At the end of n years, he owes $P(1 + i)^n$. Prepare the flowchart for a program that reads P , i (in percent), and n , and calculates the final amount owed. Run for $P = \$100$, $i = 8$ percent, and $n = 5$.

2-4. A man borrows a sum of money P . He proposes to repay this amount in n annual installments at an annual interest rate of i . The annual payment is given by

$$M = \frac{Pi(1 + i)^n}{(1 + i)^n - 1}$$

The total amount of interest he pays is

$$nM - P$$

Prepare the flowchart for a program to compute the annual payment and total interest paid if \$5,000 is borrowed at 7 percent interest for 5 years.

2-5. Prepare the flowchart for a program to evaluate the polynomial

$$f(x) = ax^3 + bx^2 + cx + d$$

for given values of a , b , c , d , and x . The number of calculations can be minimized by programming this equation as follows:

$$f(x) = ((ax + b)x + c)x + d$$

Run this program for $a = 1$, $b = 2$, $c = 7$, $d = 4$, and $x = 3$.

2-6. A cubic equation of the form $x^3 + bx^2 + cx + d = 0$ can be reduced to the form $y^3 + py + q = 0$ by the substitution $x = y - b/3$. The equations for p and q are

$$p = (3c - b^2)/3$$

$$q = (27d - 9bc + 2b^3)/27$$

Prepare the flowchart for a program to compute p and q for $b = 5$, $c = 7$, and $d = 2$. For output print the answer in equation form similar to the following:

$$Y \uparrow 3 + (-1.333)Y + (-0.407)$$

2-7. An arithmetic progression is a sequence of the type $a, a + d, a + 2d, \dots, a + (n - 1)d$ in which each term differs from the preceding term by d , called the *common difference*. The sum of n terms is given by

$$s = \frac{n}{2} [2a + (n - 1)d]$$

Prepare the flowchart for a program to compute s for $n = 20$, $a = 1.5$, and $d = 2$.

2-8. A geometric progression is a succession of terms of the type a , ar , ar^2 , . . . , ar^{n-1} , where a = first term, r = ratio of successive terms (called the *common ratio*), l = last term, n = number of terms, and s = sum of the terms. Prepare the flowchart for a program to compute l and s according to the following equations:

$$l = ar^{n-1}$$
$$s = \frac{a(r^n - 1)}{r - 1}$$

Use $a = 1.5$, $r = 0.8$, and $n = 20$.

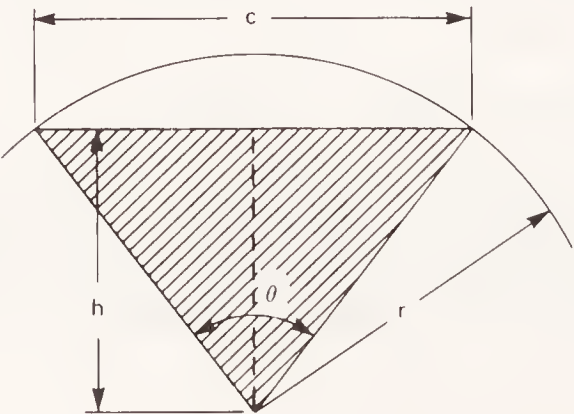
2-9. A combined arithmetic and geometric progression is of the form a , $(a + d)r$, $(a + 2d)r^2$, . . . , $[a + (n - 1)d]r^{n-1}$. The sum of n terms is given by

$$s = \frac{a - [a + (n - 1)d]r^n}{1 - r} + \frac{rd(1 - r^{n-1})}{(1 - r)^2}$$

Prepare the flowchart for a program to compute s for $a = 1.5$, $d = 2$, $r = 0.8$, and $n = 20$.

2-10. Consider preparation of a program to compute the area of the shaded region, called a *circular segment*, shown in the figure. It can be computed as follows:

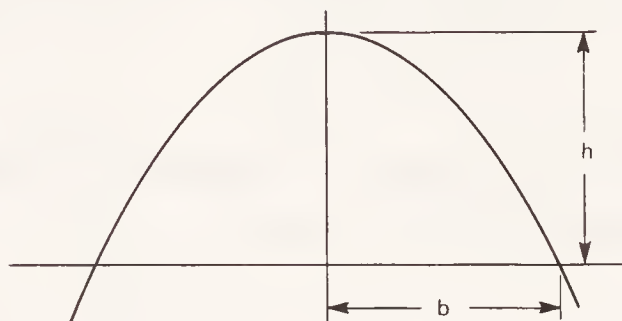
- (a) Area of cross-hatched triangle plus circular segment is the area of the circle, πr^2 , times the ratio $\theta/2\pi$, or $r^2 \theta/2$.
- (b) The length of the chord, c , is $2r \sin (\theta/2)$.
- (c) The height h of the cross-hatched triangle is $r \cos \theta/2$.
- (d) The area of the cross-hatched triangle is $hc/2$.
- (e) The area of the circular segment is therefore the area of the circular segment plus the cross-hatched triangle computed in part (a) less the area of the triangle computed in part (d).



2-11. Suppose we construct a parabola as shown in the figure. The length of the parabolic arc is given by

$$\sqrt{4h^2 + b^2} + \frac{b^2}{2h} \ln \left[\frac{2h + \sqrt{4h^2 + b^2}}{b} \right]$$

The area of the section is given by $4bh/3$. Prepare the flowchart for a program to compute the area and length of the arc for $h = 6$ and $b = 3$.



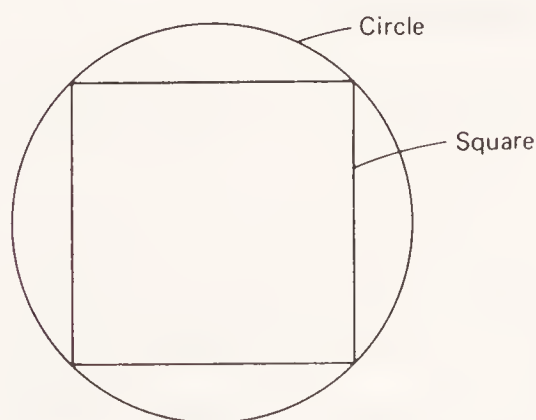
2-12. Suppose we have a circle of radius r inside of which we want to inscribe a regular polygon with n sides as illustrated in the figure for $n = 4$. The area of this polygon is given by

$$A = \frac{nr^2}{2} \sin \frac{2\pi}{n}$$

The perimeter of the inscribed polygon is given by

$$P = 2nr \sin (\pi/n)$$

Prepare the flowchart for a program to compute A and P for $n = 6$ and $r = 4$.



2-13. The hyperbolic sine and the hyperbolic cosine are given by the equations

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

Prepare the flowchart for a program to compute these two functions for $x = 2$.

ALGORITHMIC PROCESSES

70

2-14. The complex equations frequently used to give physical properties of materials as functions of other variables can be conveniently solved on the computer. For example, the thermal conductivity of oil shale is given by

$$k_s = 0.57827 (1.8081 - 0.03698 F + 0.00198 T + 0.0003056 F^2 - 0.000005184 T^2 - 0.00001872 FT)$$

where k_s = thermal conductivity, Btu/hr ft² °F/ft
 T = temperature, °C
 F = oil yield, gal/ton

Prepare the flowchart for a program to compute the thermal conductivity at 45°C of an oil shale whose oil yield is 25 gal/ton.

2-15. The vapor pressure of water between 50°C and 374.11°C is given by the following expression:

$$\log_{10} \frac{p}{p_c} = -\frac{x}{T} \left[\frac{a + bx + cx^3 + ex^4}{1 + dx} \right]$$

where p = vapor pressure, atm
 p_c = critical pressure, 218.167 atm
 T = temperature in degrees Kelvin = $t(^{\circ}\text{C}) + 273.16$
 $x = T_c - T$
 T_c = critical temperature, 647.27°K
 $a = 3.3463130$
 $b = 4.14113 \times 10^{-2}$
 $c = 7.515484 \times 10^{-9}$
 $d = 1.3794481 \times 10^{-2}$
 $e = 6.56444 \times 10^{-11}$

Prepare the flowchart for a program to compute the vapor pressure of water at 75°C.

2-16. Prepare the flowchart for a program to read a student's identification number and his marks on "test 1," "test 2," and "test 3." Also read "homework average," "midterm exam," and "final exam." Calculate "test average" for the three tests. Also calculate "composite average" based on 40 percent credit to test average, 10 percent credit to homework, 20 percent credit to the midterm exam mark, and 30 percent credit for the final exam mark. Write the student's identification number, his test average, and his composite average.

2-17. A student's grade-point average is calculated by assigning A = 4.0, B = 3.0, C = 2.0, D = 1.0, and F = 0.0 and weighing the results based on the number of hours credit (semester or quarter basis) for the course to which the grade is applied. Prepare a flowchart for a program to read a student's identification

number, his old grade-point average, and the number of hours to which this old grade-point average applied. The purpose of the program is to update this grade-point average based on the results of the five courses just completed. The program must read the hours credit and the numerical value of the grade awarded for each of these five courses. The purpose of the program is to calculate a new grade-point average for the student. Write the student's identification number, the new total number of hours to which the new grade-point average applies, and the new grade-point average itself. Do not use a loop.

2-18. Prepare the flowchart for a program entitled "Party Game." The calculations to be made are as follows:

1. Read into the machine two numbers. The first number will represent the date and month of your birth; for example, if you were born on July 10, this would be 107. The second number is to be your age at your last birthday.
2. Multiply the first number by 2.
3. Add 5 to the result.
4. Multiply the result by 50.
5. Add your age to the result.
6. Add 365 to the result.
7. Subtract 615 from the result.
8. Write the two numbers you read and the result of Step 7.

The "answer" calculated by these steps will give one number as the day and month of your birth and your current age.

2-19. A man needs a program to calculate the total taxes which he has paid during the course of a year. The final answer will indicate his relative "tax burden," that is, the ratio, expressed as a percentage, of the total taxes which he pays divided by his total income. Read his gross pay. Assume his income and social security taxes are 16 percent of his gross pay. Assume he drives an auto 10,000 miles per year, gets 15 miles per gallon, and pays 12 cents tax per gallon. Assume he pays 4 percent retail sales tax on purchases which amount to about 56 percent of his gross pay. His property tax is \$175. Assume he pays miscellaneous taxes of about 3 percent of his gross pay per year. Calculate and write his total taxes, and the tax burden ratio as defined earlier.

2-20. A company bills its customers on the last day of each month. If the customer pays his bill within the first 10 days of the next month he gets a \$10 or a 2 percent discount, whichever is more; if he pays in the next 10 days of the month, he pays the exact amount of the bill; and if he pays after this second ten-day period, he pays the bill plus a penalty of \$10 or 1 percent per month on the unpaid balance, whichever is less. As an example, if he paid his bill 25 days after the billing he would pay a \$10 or 1 percent penalty. Prepare the flowchart for a program to read the basic amount of a bill and the customer's identification number. Write his identification number, the amount of the bill if paid in

72 the first ten days, the second ten days, and the third ten days of the next month so that all of this information is available for the statement to be mailed to the customer.

2-21. Read the real part (a) and imaginary part (b) of a complex number, which is assumed to be in rectangular notation ($a + ib$) and is to be converted to trigonometric notation ($re^{i\alpha}$) by

$$\alpha = \tan^{-1} b/a$$
$$r = \sqrt{a^2 + b^2}$$

where the equivalence of ($a + ib$) and ($re^{i\alpha}$) notation is implied. Write a, b, α, r , and K for the number where

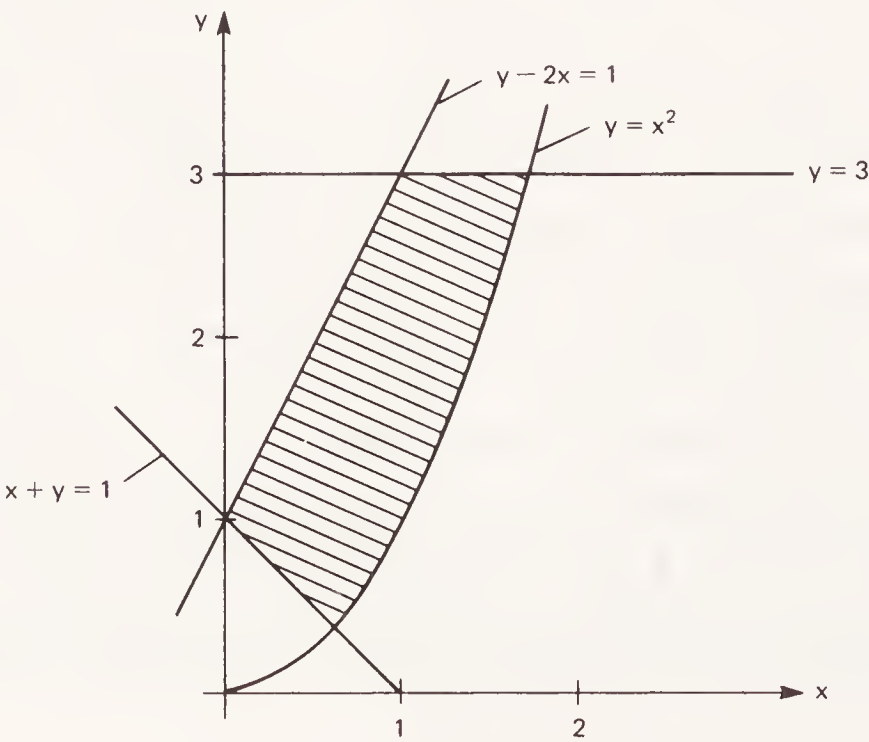
$$K = 1 \text{ if } \alpha < 0^\circ \text{ and } r < 1$$
$$K = 2 \text{ if } \alpha < 0^\circ \text{ and } r \geq 1$$
$$K = 3 \text{ if } 0^\circ \leq \alpha \text{ and } r < 1$$
$$K = 4 \text{ if } 0^\circ \leq \alpha \text{ and } r \geq 1$$

2-22. Consider a program to compute the coins to be given in change for the following situation:

- a. All purchases are under \$1.
- b. Purchaser will always present a dollar bill.
- c. Change should consist of the minimum number of coins.
- d. Only pennies, nickels, dimes, and quarters will be given in change.

The program should be written to run more than one case. Output should be in columns with amount of purchase, number of pennies, number of nickels, etc., with appropriate headings. Run the program for purchases of 97, 78, 54, 21, and 1 cent.

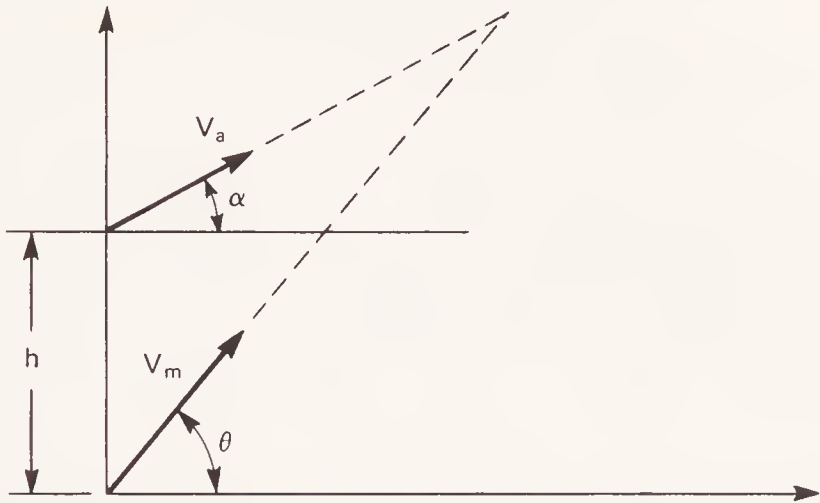
2-23. The area bounded by the inequalities



$y < 3$
 $x + y > 1$
 $y < 2x + 1$
 $y > x^2$

is illustrated in the associated figure. Prepare the flowchart for a program that accepts the coordinates of a point and determines if it is within the shaded area. The computer output should be either “yes” or “no.”

2-24. Suppose we want to fire a missile at an aircraft when it is directly overhead as illustrated in the figure. To do this, we must know the altitude h of



the plane, its velocity v_a , and its angle α with the horizontal. The angle at which to fire the missile is

$$\theta = \cos^{-1} \left[\frac{v_a \cos \alpha}{v_m} \right]$$

where v_m is the velocity of the missile. If the ratio $v_a \cos \alpha / v_m$ is greater than one, the arc cosine does not exist, which means that the aircraft is too fast for us to intercept. Furthermore, we can only aim at angles between 20 degrees and 70 degrees. The intercept time t_e is

$$t_e = \frac{h}{v_m \sin \theta - v_a \sin \alpha}$$

If t_e exceeds 10 sec, our missile fuel is expended and we cannot intercept. Prepare the flowchart for a program that, from values of v_a , h , α , and v_m , computes whether we should fire the missile. If we can make a hit, the computer should print FIRE AT ANGLE and give the angle in degrees; if not, it should print DO NOT FIRE—CANNOT INTERCEPT. Note: $\cos^{-1} x = \tan^{-1} (x / \sqrt{1 - x^2})$.

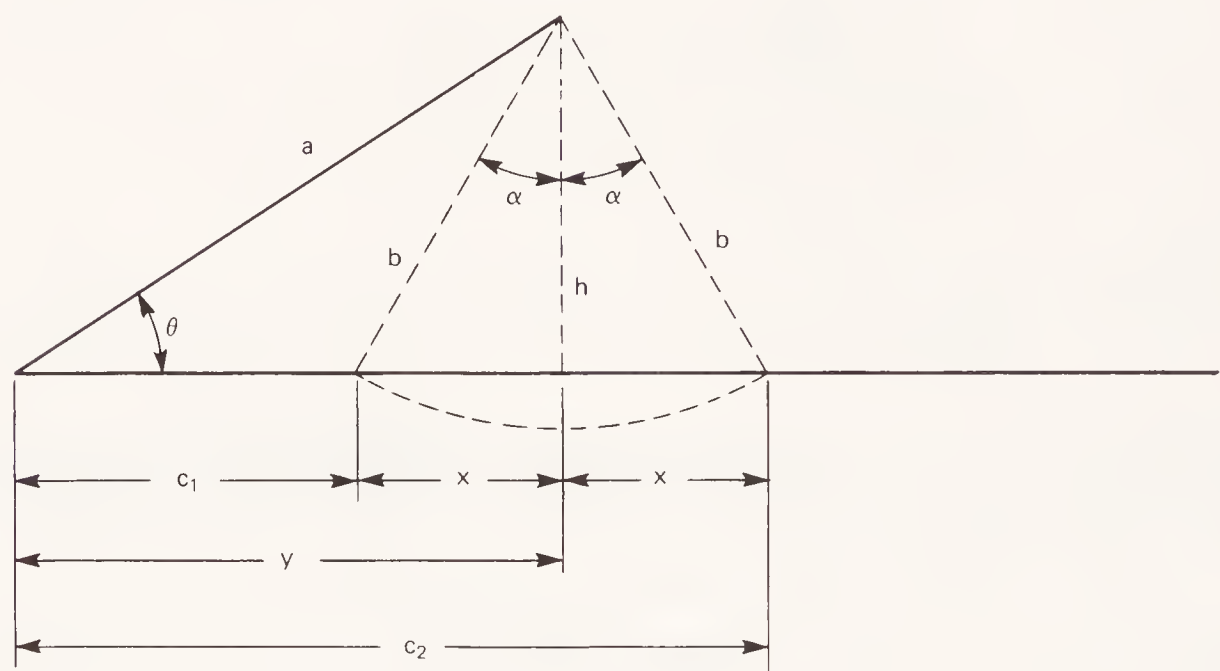
2-25. Suppose we are given an angle and two sides of a triangle. Specifically, suppose a , b , and θ in the accompanying figure are given, and we are to find the third side c . For this problem, there may be no solution, one solution, or two solutions. These can be determined as follows:

1. The angle θ and side a can be used to form a right triangle, and the sides h and y can be calculated from

$$\begin{aligned} h &= a \sin \theta \\ y &= a \cos \theta \end{aligned}$$

2. If b is less than h , no solution exists.
3. If b equals h , only one solution exists, namely, the right triangle. The third side therefore equals y .
4. If b is greater than h , the length x is computed from

$$x = \sqrt{b^2 - h^2}$$



Two further possibilities exist:

- a. If x is less than y , then two solutions exist, namely, $y + x$ and $y - x$.
- b. If x is greater than y , then only one solution exists, namely, $y + x$.

Prepare the flowchart for a program to compute the solution (or solutions) and print the results.

2-26. The solution of the simultaneous equations

$$\begin{aligned} a_1x + b_1y &= c_1 \\ a_2x + b_2y &= c_2 \end{aligned}$$

is known to be

$$\begin{aligned} x &= \frac{c_1b_2 - c_2b_1}{b_2a_1 - b_1a_2} \\ y &= \frac{a_1c_2 - a_2c_1}{b_2a_1 - b_1a_2} \end{aligned}$$

if $b_2a_1 - b_1a_2 \neq 0$. Prepare the flowchart for a program that computes x and y and prints the solution as shown in the figure. If there is no solution, the program should write “No solution.”

2-27. In Exercise 2-15, an equation was given for the vapor pressure of water from 50°C to 374.11°C. From 10°C to 150°C, the recommended equation is

$$\log_{10} \frac{p}{p_c} = - \frac{x}{T} \left[\frac{a' + b'x + c'x^3}{1 + d'x} \right]$$

where $a' = 3.2437814$
 $b' = 5.86826 \times 10^{-3}$
 $c' = 1.1702379 \times 10^{-8}$
 $d' = 2.1878462 \times 10^{-3}$

Other parameters are as defined in Exercise 2-15. Prepare the flowchart for a program to compute the vapor pressure of water using the above equation for temperatures below 100°C and the equation in Exercise 2-15 for temperatures above 100°C. Run the program for temperatures of 45°C, 90°C, and 120°C.

2-28. The reaction rate constant for the decomposition of kerogen is given by the following equations:

$$\log_{10} k = -13572/T + 18.45, \quad T < 710^\circ\text{K}$$

$$\log_{10} k = -5549/T + 7.14, \quad T \geq 710^\circ\text{K}$$

where k = reaction rate constant, min^{-1}
 T = temperature, degrees Kelvin

Prepare the flowchart for a program to compute the reaction rate constant at temperatures of 640°K, 720°K, and 810°K.

2-29. Extend Exercise 2-17 to encompass credit hours earned. In addition to the entries in Exercise 2-17, the old number of credit hours earned should be read. In updating the credit hours earned, no credit is allowed for grades of F. Write the new number of credit hours earned along with the output requested in Exercise 2-17. Do not use a loop.

2-30. Podunk U. determines an undergraduate student’s classification according to the following guidelines:

- | | |
|--------------|-----------------------------|
| 1. Freshman | less than 30 semester hours |
| 2. Sophomore | 30 to 59 semester hours |
| 3. Junior | 60 to 89 semester hours |
| 4. Senior | 90 or more semester hours |

Develop the flowchart for a program that reads the number of semester hours earned by a student and prints a 1 if a freshman, 2 if a sophomore, 3 if a junior, or 4 if a senior.

2-31. Develop the flowchart for a program that reads the time (expressed in military fashion) into a single variable and then prints the time in civilian fashion. (In effect, military time can be obtained from civilian time by simply adding 12 to the hour digits for PM times.) For example, 0845 should be printed as 8:45 AM (use the minus sign if a colon is not available). Similarly, 1829 should be printed as 6:29 PM. If the entered hour digits are greater than 24 or if the minutes digits are 60 or greater, print the message "Erroneous time entered" followed by the value entered.

2-32. This exercise entails the preparation of a program to compute the time duration in minutes of an airline flight. The time will be entered in military fashion as in the previous exercise. The first time entered will be the departure time; the second will be the arrival time at the destination. Each time should be checked for erroneous values as in the previous exercise. Furthermore, the program should function properly for flights over midnight. For example, a flight with a departure time of 2240 and an arrival time of 0120 lasts 160 minutes (no flight lasts longer than 24 hours).

2-33. Write a flowchart for a program to accept the date in the customary numerical designation and print the date with the month spelled out. That is, 2/14/72 should be written February 14, 1972. In entering the numerically designated date, the three numerical components are entered separately with the month first, the day next, and the year last. The program should also check for unreasonable entries. Specifically, the following checks should be made:

1. The numerical designation for the month should not exceed 12.
2. The entry for the day should be checked against the maximum number of days in the month. For February, the program should consider that leap years occur whenever the year is divisible by 4.
3. The year should not exceed 99.

2-34. Suppose a book company estimates the price of a book by the following guidelines: The basic cost of a book is 50 cents plus 2 cents per page, which assumes that the book will be a paperback. If, however, the number of pages exceeds 300 pages, cloth binding is used, which increases the price by \$1.00. Furthermore, if the number of pages exceeds 550, special binding procedures become necessary, which increases the cost another 75 cents. Prepare a flowchart for a program to compute the cost of a book given the number of pages.

2-35. As a follow-on to the previous exercise, suppose 800 pages is the absolute maximum number of pages that can be bound. Therefore, books containing more than 800 pages will be published in volumes of equal length but whose individual length does not exceed 550 pages. Extend the flowchart prepared for the previous exercise to encompass this situation.

2-36. In selling merchandise, a vendor uses the following procedure in computing prices:

1. The base price is applied to the first twelve units.
2. A 10 percent discount is applied to all units between 13 and 100.
3. A 30 percent discount is applied to all units above 100.

For example, suppose a purchaser orders 200 units of an item selling for \$10. The total price would be

$$\$10 \times 12 + \$9 \times 88 + \$7 \times 100 = \$1,612$$

The average price is $\$1,612/200 = \$8.06$. Prepare the flowchart for a program to read the number of units ordered and the price and compute the total price for all units and the average price.

2-37. Jalopy Rent-A-Car has decided to computerize its billing procedure. In essence, the operator enters the following information:

1. Total number of hours the customer has kept the car.
2. The daily rate for the particular car.
3. The miles driven.
4. The mileage charge in cents per mile.
5. The discount allowed for this customer.
6. An indication as to whether or not the customer took the complete collision coverage, which costs \$2.00 per day. A one indicates he took the coverage; a zero indicates he refused it.

In computing the bill, the following factors must be considered:

1. The minimum rental charge is one day's rental rate.
2. The hourly rate is 20 percent of the daily rate. Therefore, if a customer kept a car for 27 hours, he should be charged for one day plus 3 hours. However, hourly charges should not exceed the one-day rental rate.
3. If the car is kept for seven days or more, a 10 percent discount is applied to the daily rental rate (but not the mileage rate).
4. The maximum insurance rate is \$10 per week.
5. The customer discount is applied to all but the insurance rate.
6. A 6 percent sales tax should be added to the bill.

Prepare the flowchart for this program.

2-38. The process of raising a real number to a power occasionally presents problems. When the power contains a fractional part, this operation is computed as follows:

$$a^b = 10^{b \cdot \log_{10} a} \text{ or } \exp(b \cdot \ln a)$$

Problems arise because the logarithm of a negative number does not exist, and therefore a is restricted to positive values. In some cases, however, this approach does not lead to the correct answer. For example, the cube root of -27 is -3 .

78 But programming as $(-27) \uparrow 0.3333$ causes the computer to attempt to take the logarithm of a negative number. Can you devise an expression by which the cube root can be computed and this problem avoided?

2-39. Via a method presented in *Puzzles and Paradoxes* by T. H. O’Beirne (Oxford University Press, London, © 1965), the date for Easter can be computed for any year x by the following ten steps:

Step	Dividend	Divisor	Quotient	Remainder
1	x	19	—	a
2	x	100	b	c
3	b	4	d	e
4	$8b + 13$	25	g	—
5	$19a + b - d - g + 15$	30	—	h
6	$a + 11h$	319	μ	—
7	c	4	i	k
8	$2e + 2i - k - h + \mu + 32$	7	—	λ
9	$h - \mu + \lambda + 90$	25	n	—
10	$h - \mu + \lambda + n + 19$	32	—	p

The month in which Easter occurs is given by n and the day of the month by p . For example, using 1972 for x gives a value of 4 and 2 for n and p , respectively, which corresponds to a date of April 2 for Easter. Prepare the flowchart for a program to perform these computations.

2-40. As an alternate to the procedure given in the previous exercise, O’Beirne proposes the following:

Step	Dividend	Divisor	Quotient	Remainder
1	x	100	b	c
2	$5b + c$	19	—	a
3	$3(b + 25)$	4	δ	ϵ
4	$8(b + 11)$	25	γ	—
5	$19a + \delta - \gamma$	30	—	h
6	$a + 11h$	319	μ	—
7	$60(5 - \epsilon) + c$	4	j	k
8	$2j - k - h + \mu$	7	—	λ
9	$h - \mu + \lambda + 110$	30	n	q
10	$q + 5 - n$	(32)	(0)	p

Prepare the flowchart for a program to perform these computations.

3

LOOPS AND ITERATION COUNTERS

A significant component in the measure of one's programming skill is the ability to use loops effectively. In the last chapter, we noted the existence of loops in our flowcharts, and even presented an example in which a loop was used. Virtually this entire chapter is devoted to the use of loops. The best way to learn how to use loops is by studying examples involving their uses, an orientation we shall adopt throughout this chapter. After study of these examples, the reader should be able to detect when loops should be used in the subsequent problems he encounters.

3.1. CONSTRUCTION OF SIMPLE LOOPS

The loop is a common and extremely useful feature in programming. In this section, we shall examine several techniques by which a loop can be constructed to compute the average of a set of numbers. Specifically, suppose the numbers comprising this set are

1.2, 5.7, -1.4, 2.6, -0.2

The sum of these numbers is 7.9, and the average is $7.9/5$, or 1.58. How can we get a computer to do this?

One possibility is illustrated by the flowchart in Figure 3-1. We simply read five numbers, compute their sum, and divide the result by 5. But what if the set consists of a thousand numbers? While this approach could be used, it would not be very convenient. But even further, what if the number of entries in the set varies from one application to the next? Do we need a separate program corresponding to each conceivable number of entries in the data set? As we shall see, use of a loop avoids both of these difficulties.

How do you construct a loop to compute the sum of a set of numbers? One approach is to establish a storage location for the sum and proceed as follows:

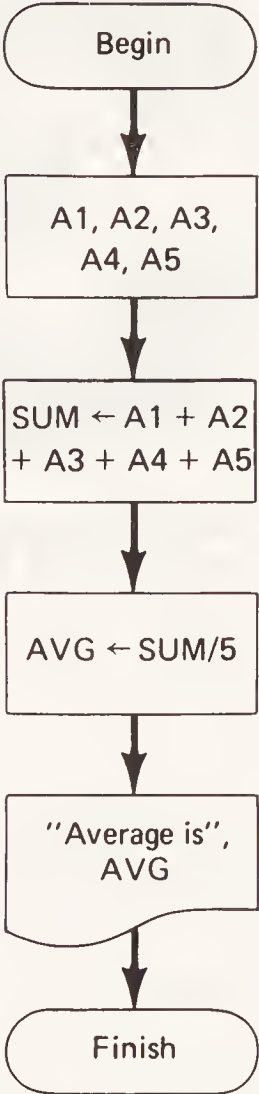


Figure 3-1 Computation of the average of five numbers

- Step 1. Initialize the sum to zero.
- Step 2. Read a number.
- Step 3. Increase the value of the sum by this number.
- Step 4. Return to Step 2.

This is illustrated by the flowchart in Figure 3-2. An output block has also been inserted so that the progression of the computations can be followed. For the set of numbers given previously, the output would be as follows:

A	SUM
1.2	1.2
5.7	6.9
- 1.4	5.5
2.6	8.1
- 0.2	7.9

One interesting feature of the flowchart in Figure 3-2 is that there is no “Finish” block. In essence, the computer would continue the computations as long as more data are available.

In order to compute the average, an exit from this loop must be provided. One approach is to add after the last true entry a “trailer” that can be

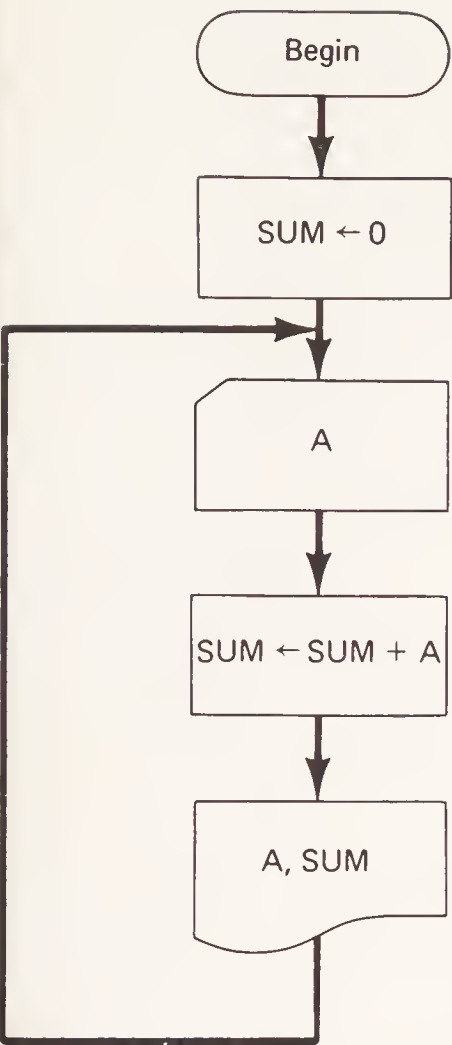


Figure 3-2 Loop for summing numbers

detected by a decision block. As one example, suppose the data set is known not to contain a zero entry. Then a “trailer” with a zero entry can be added to the data set and a decision block inserted immediately after the input block to detect this entry and transfer control outside the loop. The flowchart in Figure 3-3 illustrates this approach.

Of course, some data sets may contain a zero entry. For those that may contain a zero entry but not a negative entry, the program can be written to detect a trailer with a negative entry. When the data may consist of positive, negative, and zero entries, an extremely large positive or negative number may be used. For example, the decision block may check for a trailer with an entry such as 10^{50} , which it is hoped would not be normally encountered in the data set.

As presented in Figure 3-3, the program would be restricted to cases where there are exactly five numbers in the set. This restriction could be removed by reading a value for the number of entries and dividing by this number when computing the average. Especially when we deal with large sets of data, a more convenient approach would be to let the computer count the entries as they are processed, as illustrated in Figure 3-4. This same program could be used for five entries, for a thousand entries, or however many are provided. It is completely independent of the number of entries in the data set.

Some programming languages (for example, Fortran and Cobol) provide an AT END option in their READ statements that detect the end of data. As

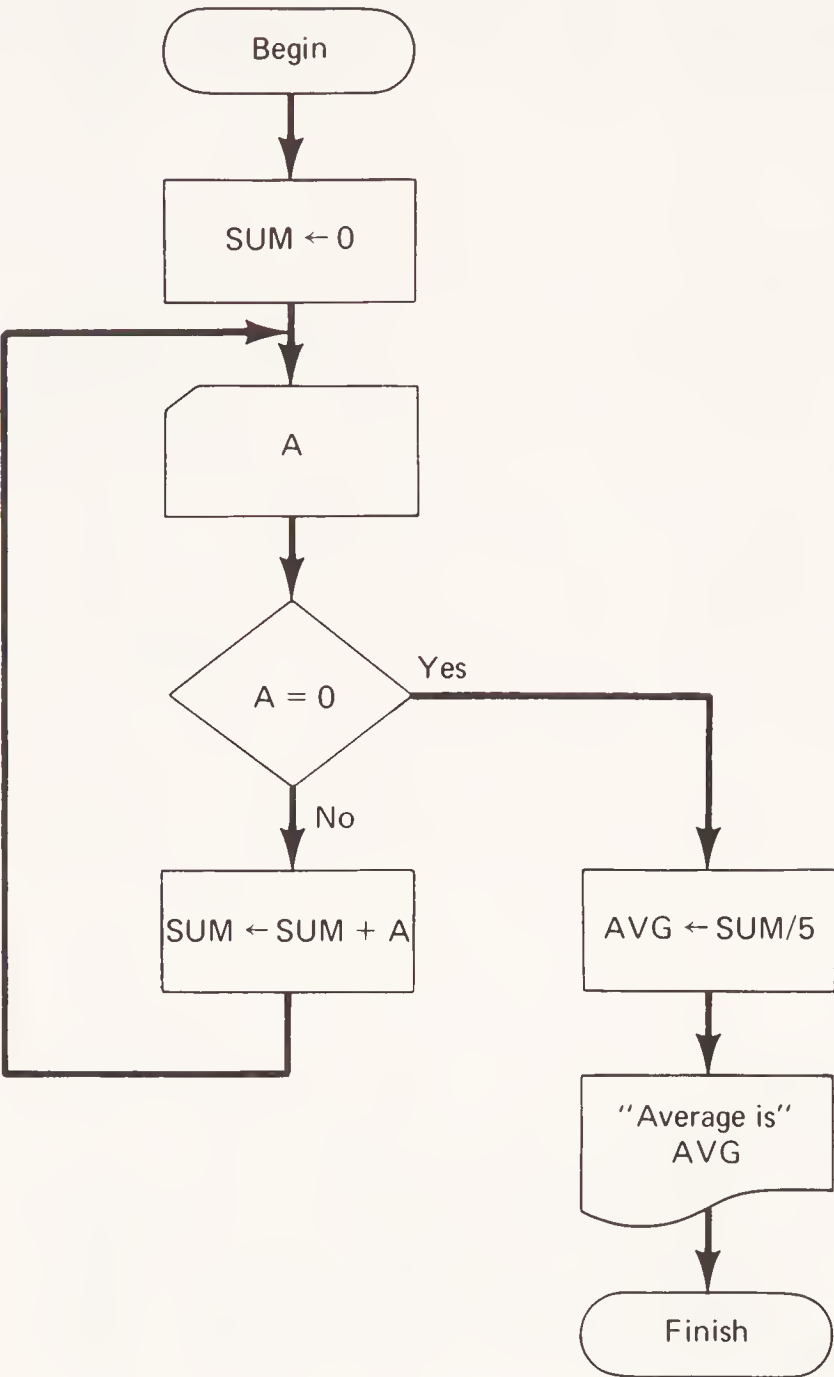
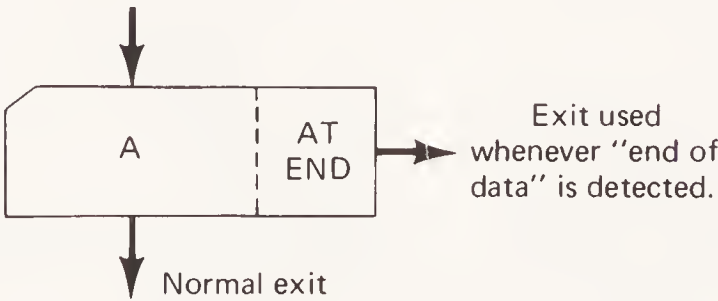


Figure 3-3 Flowchart for computing average using a zero entry to indicate "end of data"

illustrated in Figure 1-14, a control card follows the last data entry in the deck prepared for a batch processing system. Upon encounter of this control card, the AT END option directs the computer to a statement within the user's program. In our flowchart examples, we shall use an AT END box to the right of the input block, as illustrated below:



Using this feature, the flowchart for computing the average is illustrated in Figure 3-5.

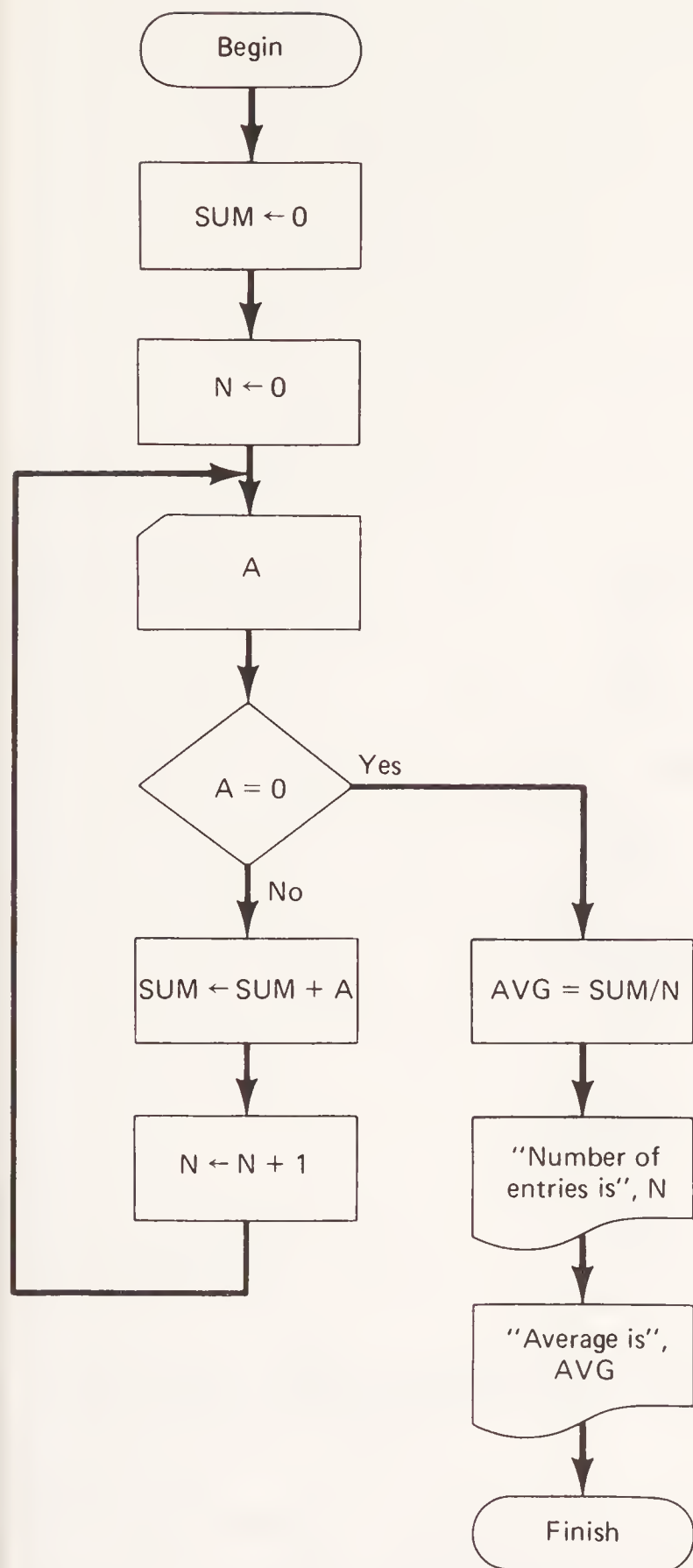


Figure 3-4 Flowchart for program in which the number of values is not known beforehand

3.2. ITERATION COUNTERS

For the example in the previous section, suppose the number of entries is supplied to the program on the first data card. Then we might modify the program in Figure 3-5 by deleting the decision block to detect the trailer card and inserting one to check the number of entries processed, as illustrated in

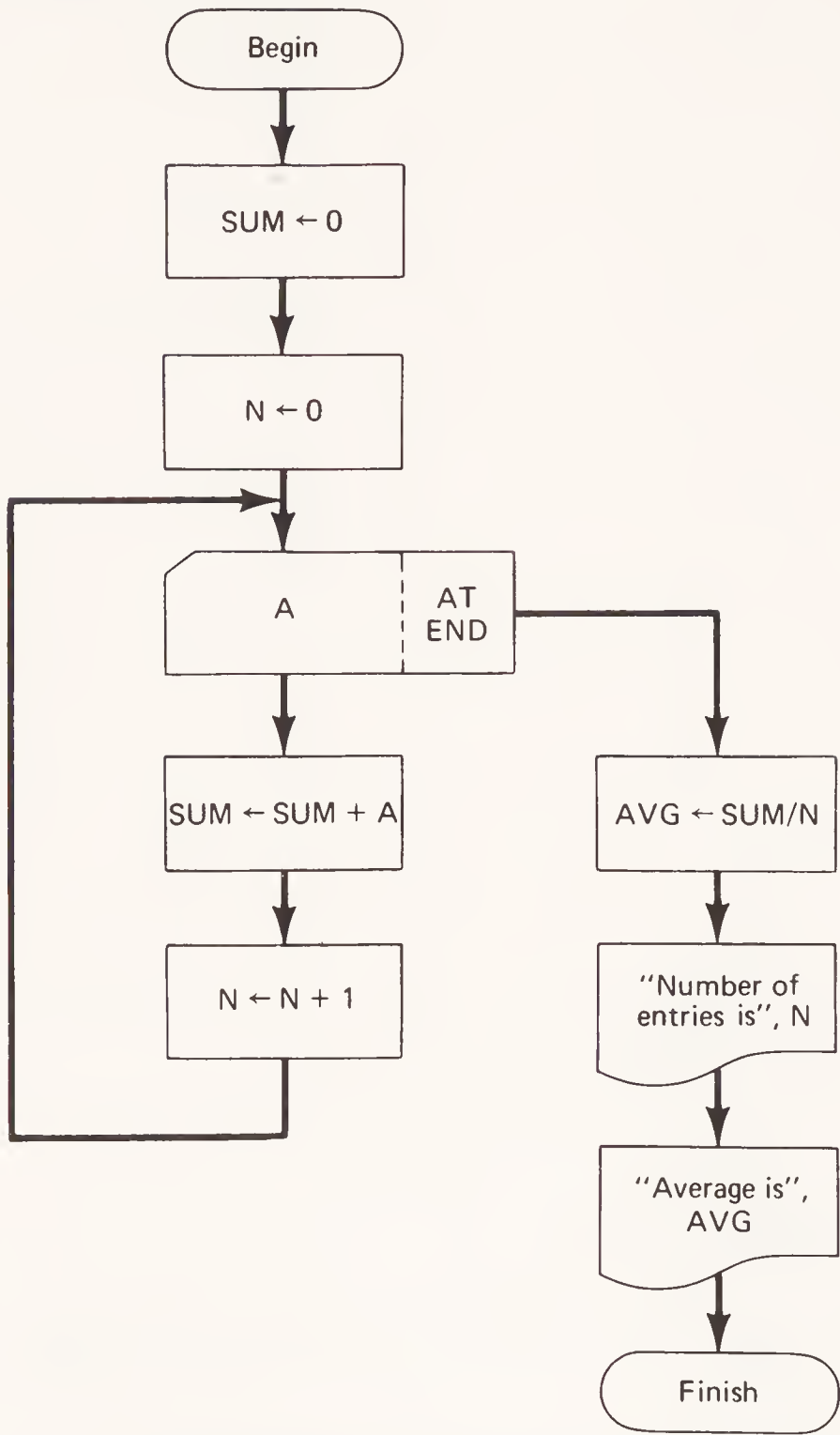


Figure 3-5 Flowchart of a program that counts the entries and computes the average

Figure 3-6. By this procedure we have constructed an iteration loop that is executed a specified number of times under control of the counter.

Although an iteration counter can be readily constructed using a decision block as in Figure 3-6, most programming languages provide special statements that make programming of iteration counters even more convenient.

In fact, an iteration counter can be constructed in ways other than shown in Figure 3-6. Suppose we want to construct an iteration counter around a group of statements using INDEX as the index of our counter. On the first iteration, we want INDEX to equal INIT. We want to increase the value of INDEX by INCRA after each execution of the statements and continue until INDEX exceeds LIMIT.

Figure 3-7 shows two ways to construct the iteration counter. In many respects, these loops are equivalent. There is, however, one important point of

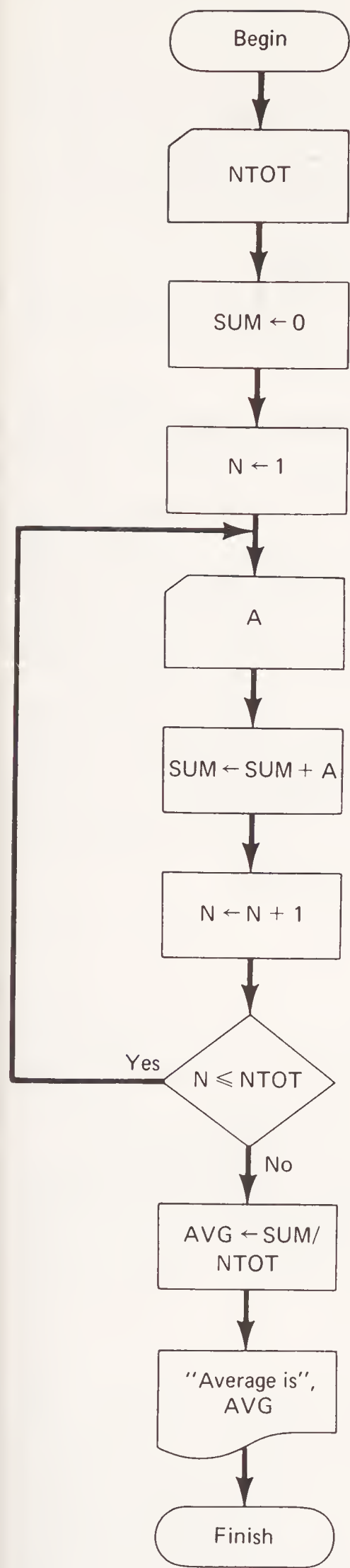


Figure 3-6 Flowchart with a counter

distinction which should be noted. Suppose INIT exceeds LIMIT. In version 1 of the iteration counter in Figure 3-7, the statements in the loop would still be executed *once*, since the INDEX is not compared to LIMIT until *after* the

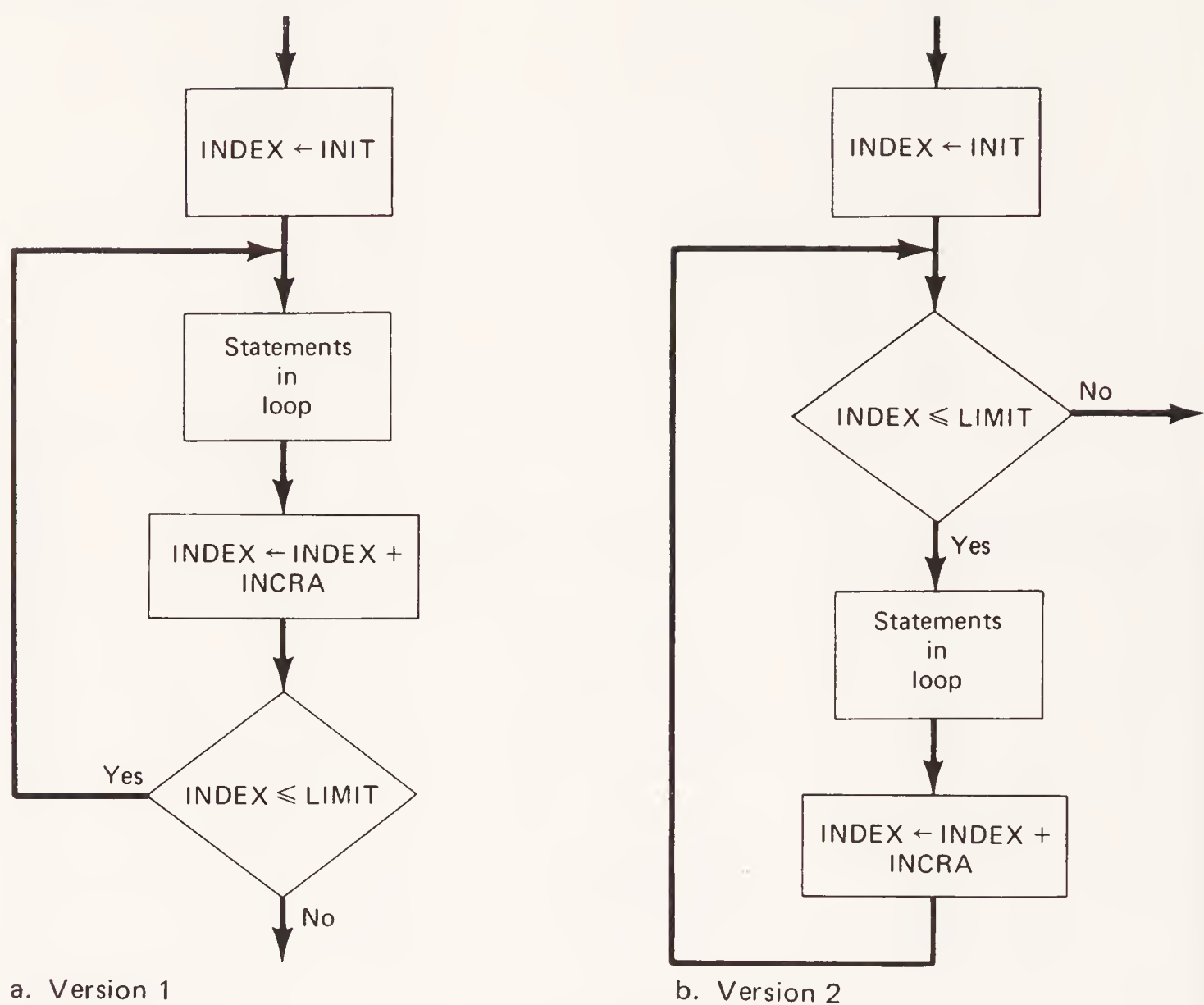


Figure 3-7 Construction of iteration counters

statements have been executed. But in version 2 in Figure 3-7, the statements are *not* executed since INDEX is compared to LIMIT *before* the statements are executed. When INIT exceeds LIMIT and version 2 of the iteration counter is used, we are said to have a *vacuous loop*, that is, one that is not executed.

What is the value of INDEX when an exit is made from either of the loops in Figure 3-7? Our intuition tells us that it should equal LIMIT, but it is not. Suppose INIT equals 1, INCRA equals 1, and LIMIT equals 5. Then INDEX assumes the sequence of values

$$\text{INDEX} = 1, 2, 3, 4, 5, 6$$

and the exit is made when the value of INDEX equals 6. However, the statements in the loop are executed only for INDEX equal to 1, 2, 3, 4, and 5, or a total of five iterations. In this case, INDEX equals LIMIT plus INCRA after exiting from the loop, but this is not always true. For example, suppose INIT equals 1, LIMIT equals 6, and INCRA equals 2. Now INDEX assumes the values

$$N = 1, 3, 5, 7$$

The value of LIMIT is never assumed, the exit occurring the first time INDEX

assumes a value that exceeds LIMIT. In this case, the statements in the loop are executed for INDEX equal to 1, 3, 5.

Let's return to the question of the value assumed by N after exit from the loop. The following expression applies to both iteration counters in Figure 3-7:

$$\text{INDEX} = \text{INIT} + \text{INCRA} \times \text{INT}((\text{LIMIT} - \text{INIT})/\text{INCRA} + 1)$$

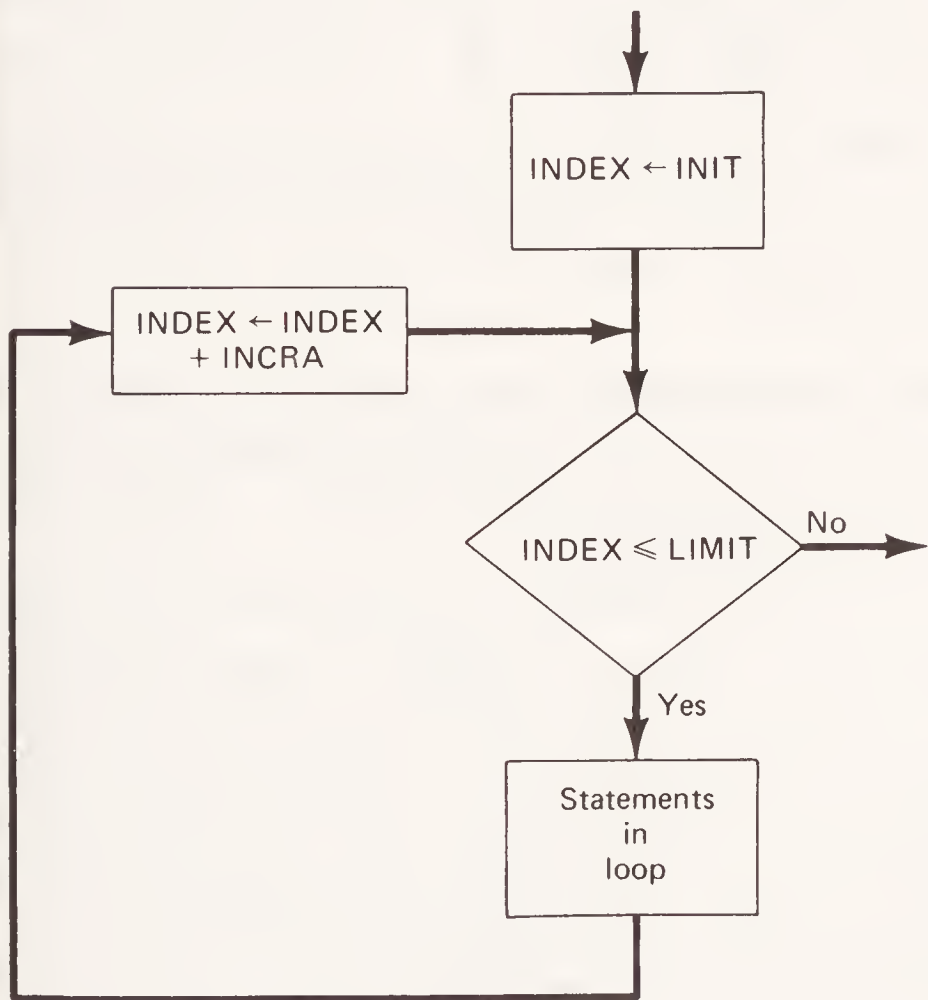
The number of iterations is given by

$$\text{INT}((\text{LIMIT} - \text{INIT})/\text{INCRA}) + 1$$

These expressions assume that the values of the *indexing parameters* INIT, INCRA, and LIMIT do not change during execution of the loop.

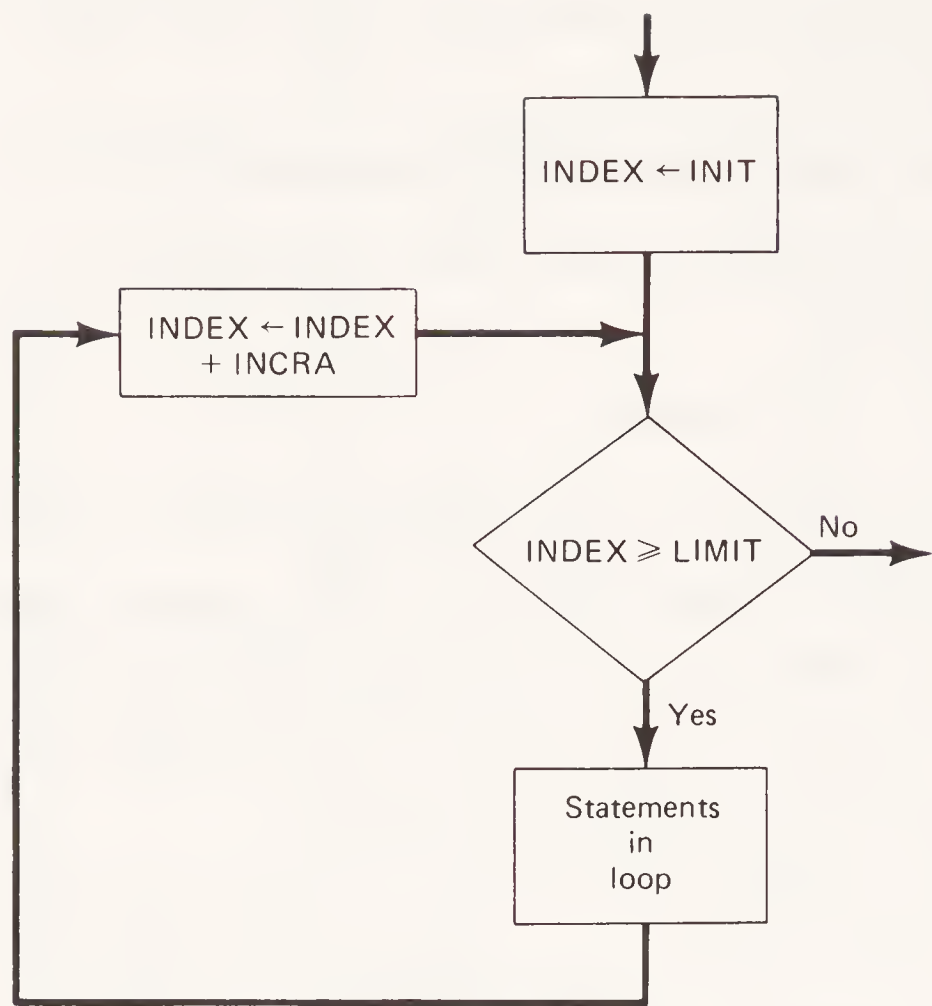
Although the iteration counter is used sufficiently often in programming languages to warrant a special statement, we shall use a combination of the blocks already introduced rather than devise a special block for our flowcharts. Choosing version 2 of the iteration counter in Figure 3-7 as the basis, our representation of the iteration counter will appear as in Figure 3-8.

Note that there is a version for positive incrementing ($\text{INCRA} > 0$) and a version for negative incrementing ($\text{INCRA} < 0$), the only difference being that the relational operator $\leq$ is used for positive incrementing, whereas $\geq$ is used for negative incrementing. Except that a zero increment ($\text{INCRA} = 0$) is not reasonable, no other restrictions are made on the index or the indexing



a. Positive incrementing (INCRA > 0)

Figure 3-8 Iteration counter blocks



b. Negative incrementing (INCRA < 0)
Figure 3-8 (Continued)

parameters INIT, INCRA, and LIMIT. Positive and negative values are accepted and they are *not* restricted to integer values; that is, fractional parts are permitted.

In general, we shall also permit expressions to be used for the parameters INIT, INCRA, and LIMIT. The expressions may involve variables whose values change during execution of the loop. Let us note the following points about the use of expressions in iteration counters:

1. The expression for INIT is evaluated only once, and prior to execution of any statements in the loop. Therefore, even if the variables involved in the expression for INIT are changed during execution of the loop, the initial value of the index is not affected.
2. The expressions for INCRA and LIMIT are re-evaluated on each pass around the loop. Therefore, changes in variables in these expressions affect the indexing. If, however, these expressions do *not* contain variables whose values will change, the repeated computation of these expressions is effectively wasting the computer's effort. The expressions should be precomputed and stored in variables that are then used as the indexing parameters. This applies only to INCRA and LIMIT; there is no incentive to precompute the expression for INIT.

In concluding this section, we should point out that some but not all programming languages permit this much generality in their special statements for iteration counters. Some or all of the following restrictions are often made:

1. The index and indexing parameters must be integers.
2. Only positive increments can be used.
3. The index is restricted to only positive values.
4. No expressions are permitted for the indexing parameters.
5. The values of neither the index nor the indexing parameters may be changed during execution of the statements in the loop.
6. Upon leaving the loop via the exit in which INDEX is compared with LIMIT, INDEX is thereafter considered to be undefined.

3.3 EXAMPLES OF THE USE OF LOOPS

Example 1. Fibonacci Sequence

Suppose we begin with the two numbers 0 and 1, which we will consider as entry 1 and entry 2 in our sequence. If entry 3 is the sum of entries 1 and 2, its value will be 1. If entry 4 is the sum of entries 2 and 3, its value will be 2. Continuing this process gives the Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Our problem is to construct an algorithm to compute the value of a desired entry in the sequence.

Note that in order to compute an entry, only the two previous entries must be known. Suppose we let ENT1 be the first previous entry and ENT2 be the second previous entry. The first step in our algorithm is to initialize the values of ENT1 and ENT2 as follows:

ENT1 $\leftarrow$ 1
ENT2 $\leftarrow$ 0

The next step is to read the number of the entry to be computed, say, NENT.

At this point, we are ready to enter a loop to compute successive values until entry NENT is reached. As illustrated by the flowchart in Figure 3-9, the loop counter J is initialized at 3, since the first two entries are known. It is incremented by 1 until entry NENT is reached.

On each pass through the loop, the value of the current entry, ENT, is computed as the sum of the previous two. Then the values of the previous two entries must be updated so that the loop can be repeated. The value of the first previous entry is stored in the location for the second previous entry. The value of the current entry is stored in the location for the first previous entry. To illustrate this process, suppose we insert an output statement immediately following the computation of the current entry ENT in the flowchart in Figure 3-9. The output would be as shown in Table 3-1.

The order of the statements to update the two previous entries at the end of the loop is critical. Consider the following sequence:

Table 3-1 Progression of the Computation of the Fibonacci Numbers

J	ENT1	ENT2	ENT
3	0	1	1
4	1	1	2
5	1	2	3
6	2	3	5
7	3	5	8
8	5	8	13
9	8	13	21
10	13	21	34

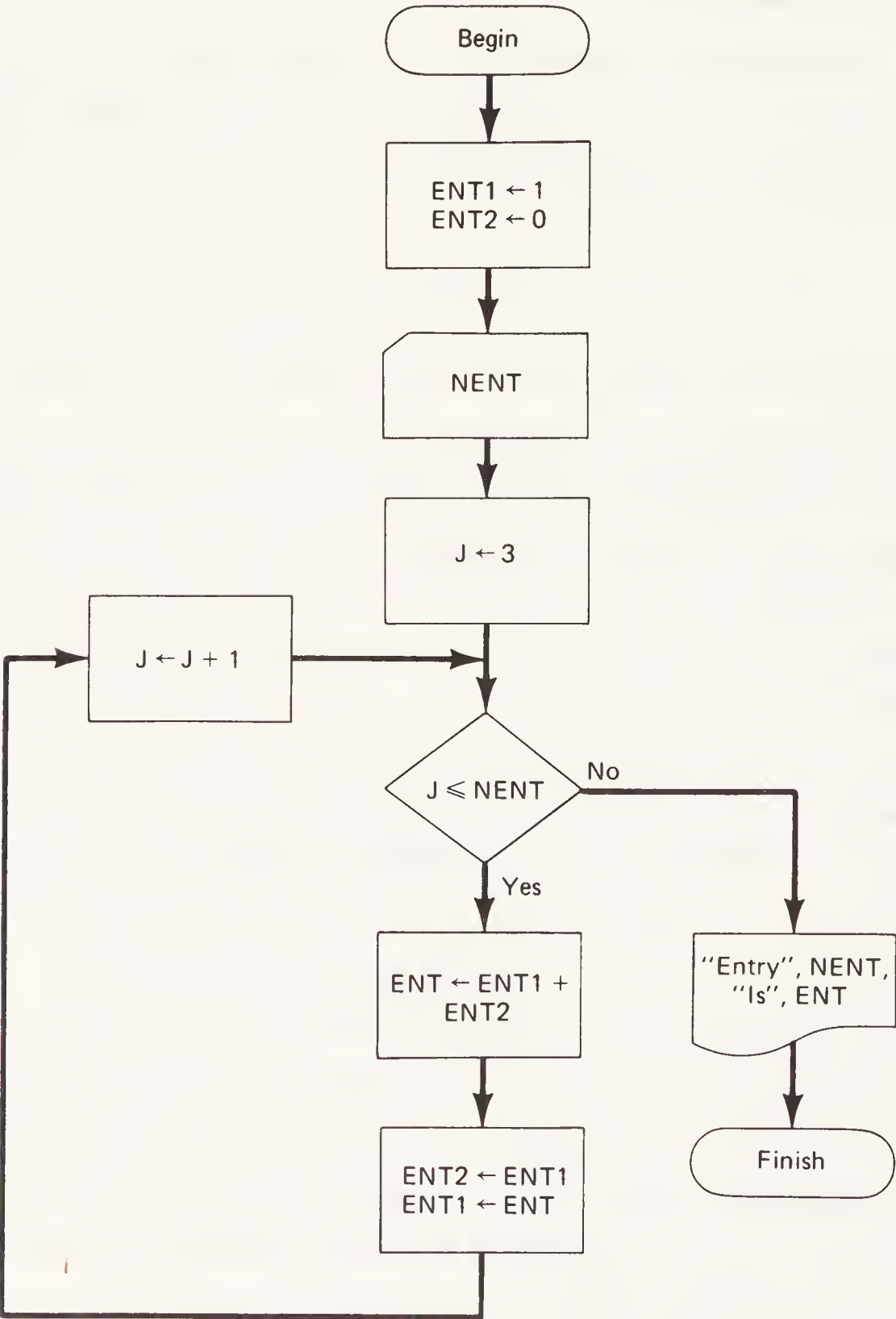


Figure 3-9 Flowchart of the algorithm to compute the Fibonacci sequence

```
ENT ← 8
ENT1 ← 5
ENT2 ← 3
ENT2 ← ENT1
ENT1 ← ENT
```

The resulting values for ENT1 and ENT2 are 8 and 5, respectively, which is correct. Suppose the last two statements are interchanged:

```
ENT ← 8
ENT1 ← 5
ENT2 ← 3
ENT1 ← ENT
ENT2 ← ENT1
```

The resulting values for ENT1 and ENT2 are both 8, which is incorrect.

Example 2. Depreciation via the Sum of the Years' Digits Method

Suppose a man makes an investment of \$30,000 in a piece of capital equipment. For tax purposes, he is permitted to depreciate his investment over the useful life of the equipment, say 5 years for our example. In order to recover an investment as quickly as possible, it is desirable to depreciate the largest amount permitted in the first year and less each succeeding year. One formula for doing this is called the *sum of the years' digits method*.

Computation of depreciation by this method is illustrated in Table 3-2. The first step is to compute the sum of the years' digits, in our case the sum of the numbers 1 through 5. This can be done by an iteration counter or alternatively noting that the sum of the integer numbers from 1 to n equals $n \cdot (n + 1)/2$. In our example, we shall use the former approach.

The depreciation for each year is computed as a fraction times the original investment. The numerator of the fraction consists of the years' digits used in *reverse* order. The denominator is the sum-of-the-years' digits. The book value at the end of the year is the book value at the end of the previous year (or for the first year, the original investment) less the depreciation for the year.

Table 3-2 Depreciation via the Sum of the Years' Digits Method

Year	Depreciation	Book value
0	0	\$30,000
1	$5/15 \times \$30,000 = \$10,000$	\$20,000
2	$4/15 \times \$30,000 = \$8,000$	\$12,000
3	$3/15 \times \$30,000 = \$6,000$	\$6,000
4	$2/15 \times \$30,000 = \$4,000$	\$2,000
5	$1/15 \times \$30,000 = \$2,000$	0
15 = sum of the years' digits		

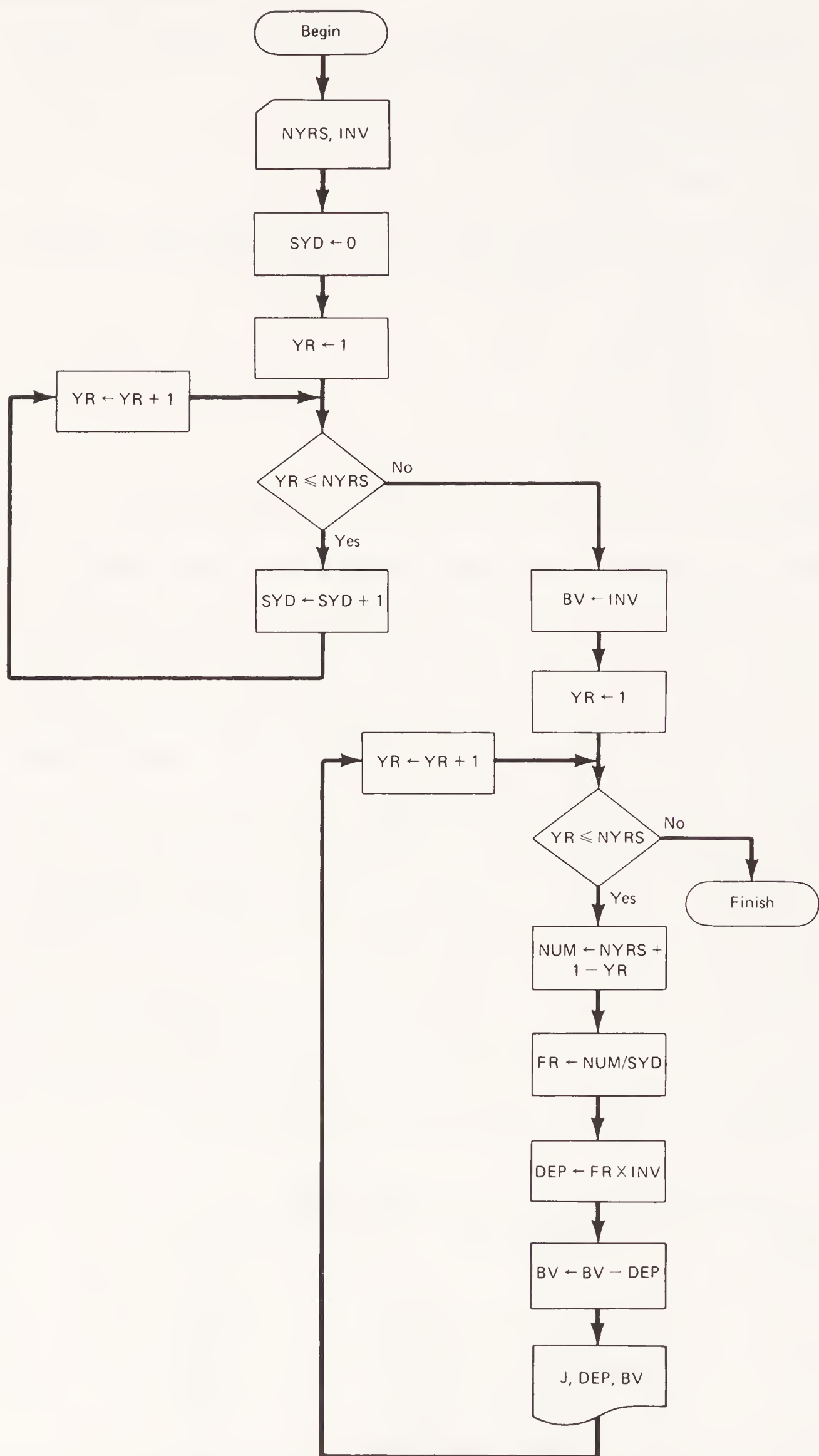


Figure 3-10 Flowchart for computation of depreciation via the sum of the years' digits method

The flowchart for the algorithm to compute the depreciation via this method is given in Figure 3-10. This program contains two iteration counters—one to compute the sum of the years' digits (SYD) and one to compute the depreciation (DEP) for each year and subtract it from the book value (BV). In this case, the loops are completely independent of each other.

3.4. NESTED LOOPS

The examples in the previous sections contained no loops inside another loop, or, in computer terminology, nested loops. By using nested loops, either with or without an iteration counter, the programmer can instruct the computer to undertake quite intricate computational procedures.

As an example, suppose we would like to determine all of the factors of an integer number. This is really an extension of our prime number algorithm, which had to determine only the first factor. For example, the number 10,340 has the factors 2, 2, 5, 11, and 47. To determine these factors, we begin by dividing by 2. If the result is an integer, then 2 is a factor. But since 2 may be a repeated factor, we divide the resulting integer by 2 and continue as long as the result of the division is an integer. As soon as we divide by 2 and obtain a noninteger result, there are no more repeated factors. We then repeat the same procedure for 3, again checking for the possibility of repeated factors. Then we try 4, then 5, etc., until we divide by the number itself, obtaining the result of 1.

To illustrate this procedure on a specific example, consider finding the factors of 10,340. First, we divide 10,340 by 2, obtaining the result 5,170. Since this is an integer, 2 is a factor. We must try 2 again. Dividing 5,170 by 2 yields 2,585. As this is also an integer, 2 is a repeated factor. Since dividing 2,585 by 2 gives the noninteger 1,292.5, we then divide 2,585 by 3, obtaining 861.67. Since this is not an integer, 3 is not a factor. Dividing by 4 also does not give an integer result. Dividing by 5 does, however, the result being 517. Dividing by 5 again indicates that 5 is not a repeated factor. We then try 6, then 7, 8, 9, and 10. Dividing by 11 gives an integer result, namely, 47. We try 11 again, obtaining a noninteger result. We then try 12, 13, etc., until we get to 47, the last factor.

Figure 3-11 gives the flowchart of our factoring algorithm. This flowchart contains two loops, one within another. The inner loop tests for repeated factors and is not controlled by an iteration counter. This loop is simply repeated as many times as is necessary until all repeated factors are located.

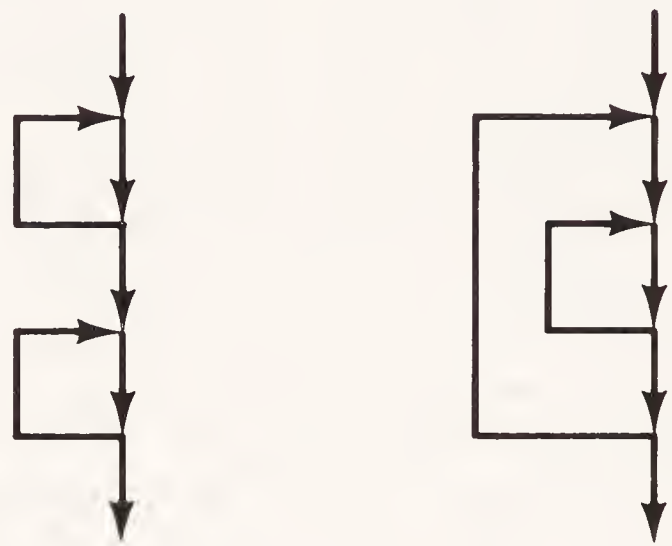
Although the outer loop in the flowchart is controlled by an iteration counter, it can be programmed quite conveniently without it, as illustrated in Figure 3-12. It is permissible, however, for both inner and outer loops to be controlled by an iteration counter.

As with the prime-number algorithm considered in the last chapter, certain improvements can be made in our algorithm. First, after all factors of 2 have been found, only odd integers need be considered thereafter. Also, the iteration counter in Figure 3-11 can be terminated at $\text{SQR}(\text{NUMBER})$ instead of NUM-

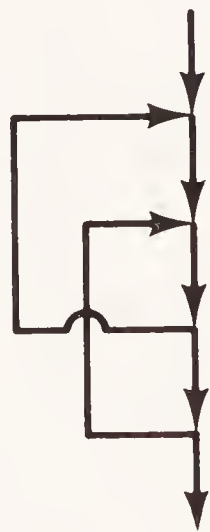
94 BER. The value of NUMBER must, however, be printed as a factor unless its value is 1.

 If we would like to find the factors of all integer numbers from 5,780 to 5,790, another loop controlled by an iteration counter can be superimposed over the flowchart in either Figure 3-11 or 3-12. Using Figure 3-11 as a starting point, we find the result shown in Figure 3-13.

 The arrangement of nested loops must be such that one loop lies either entirely within or entirely outside another loop. That is, either of the following arrangements is acceptable:



The right-hand example contains nested loops; the left-hand example contains two independent loops. The following arrangement is not acceptable:



The two loops overlap and will not function properly.

 To illustrate the problems with this arrangement, consider the flowchart in Figure 3-14. The first loop should vary the index J from 1 to 3 in increments of 1. The second loop should vary its index K from 1 to 2. Tracing the flow of execution will reveal the following output:

J = 1 K = 1
J = 2 K = 1
J = 3 K = 1
J – LOOP SATISFIED

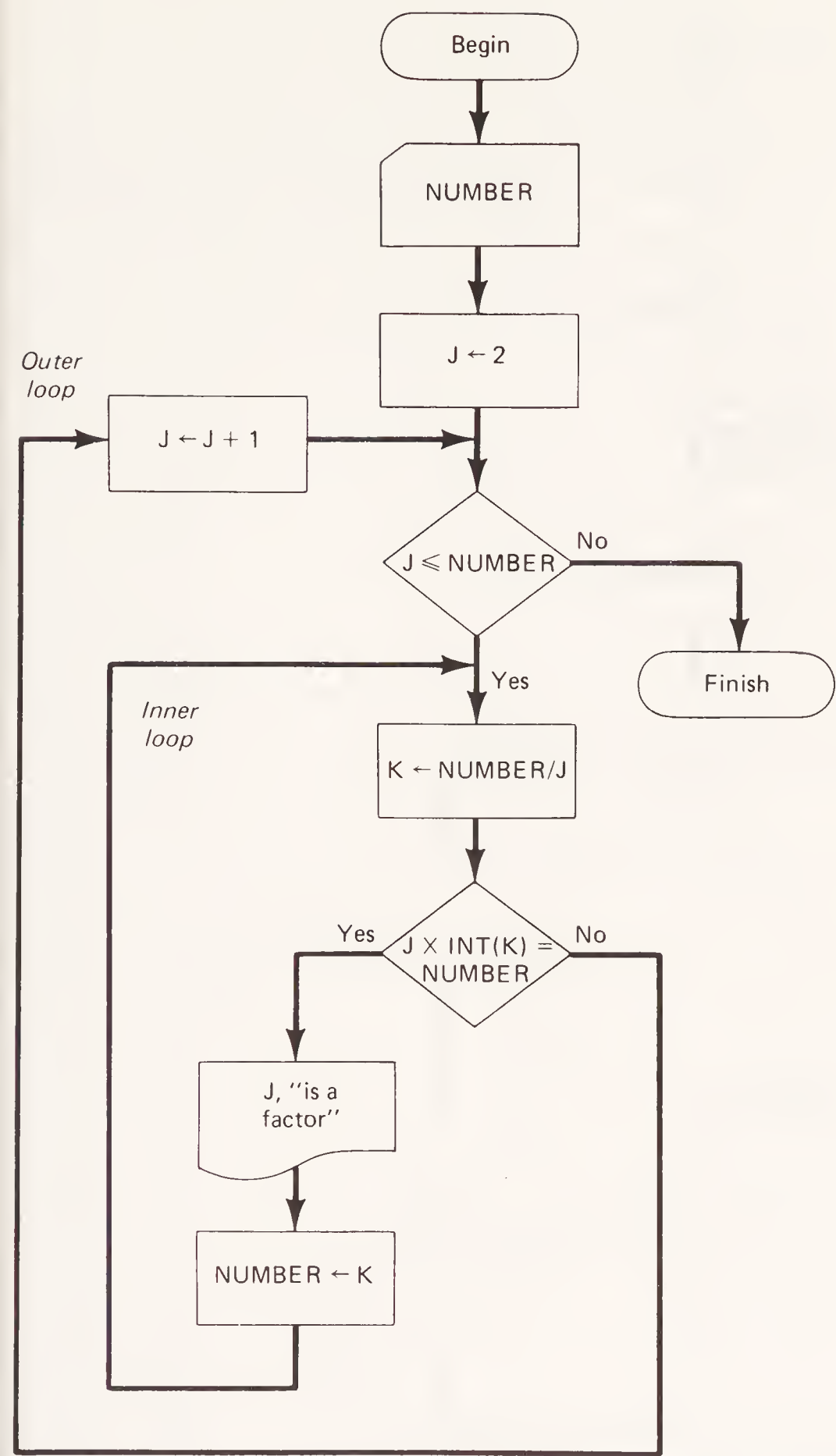


Figure 3-11 Flowchart of the factoring algorithm

J = 4 K = 2
J – LOOP SATISFIED
K – LOOP SATISFIED

The basic problem is that the statements common to both loops will be executed with a value of the index of the first loop greater than its limit. Note in this example that the print block is executed with J equal to 4.

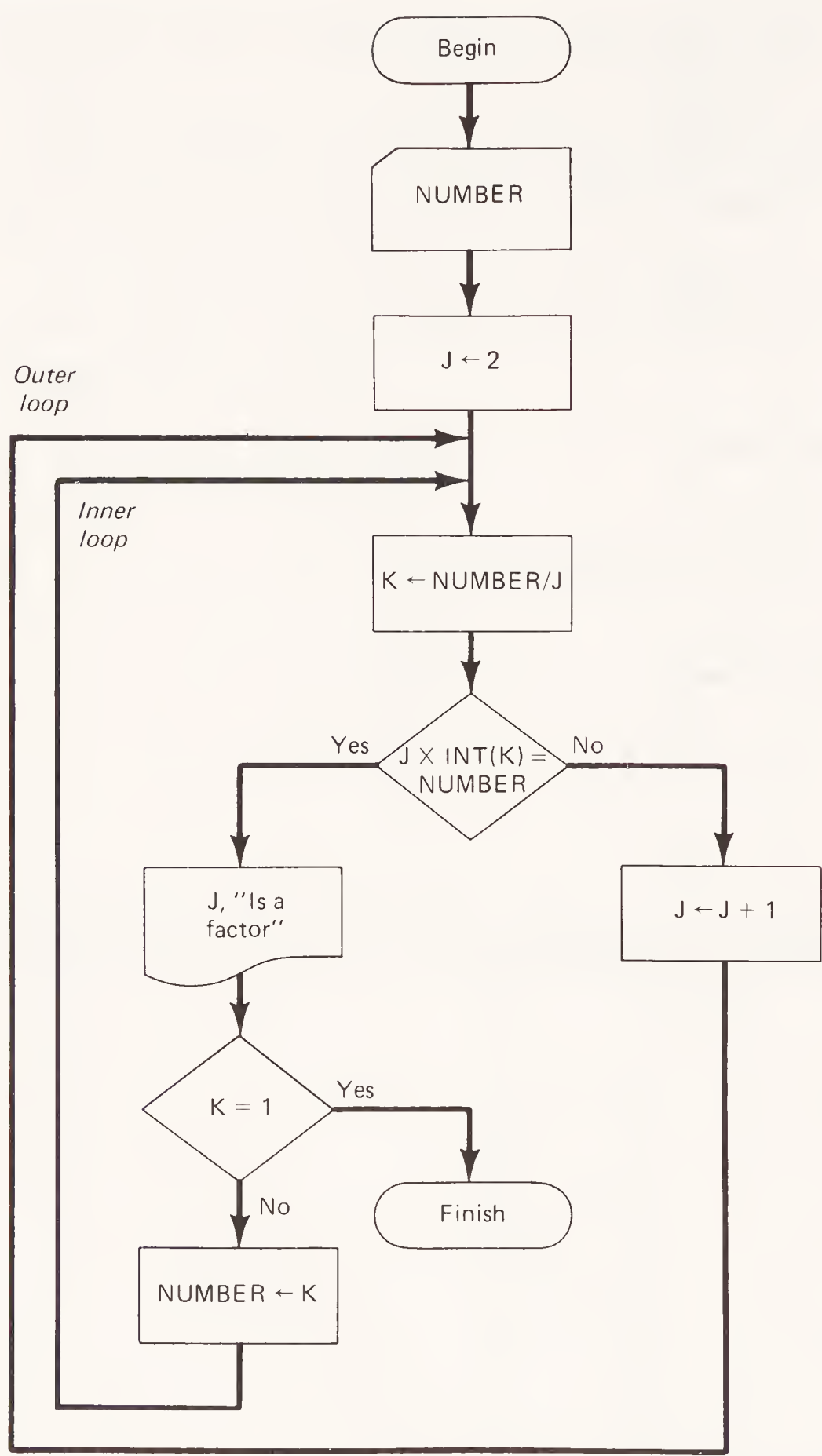


Figure 3-12 Flowchart of the factoring algorithm that contains no iteration counter

3.5. DEBUGGING

By now, those students who have been conscientiously doing their homework have discovered a phenomenon known in computer terminology as a “bug”—a flaw in the program that causes the computer to give either error messages, incorrect answers, or sometimes no answers at all. Bugs that generate error

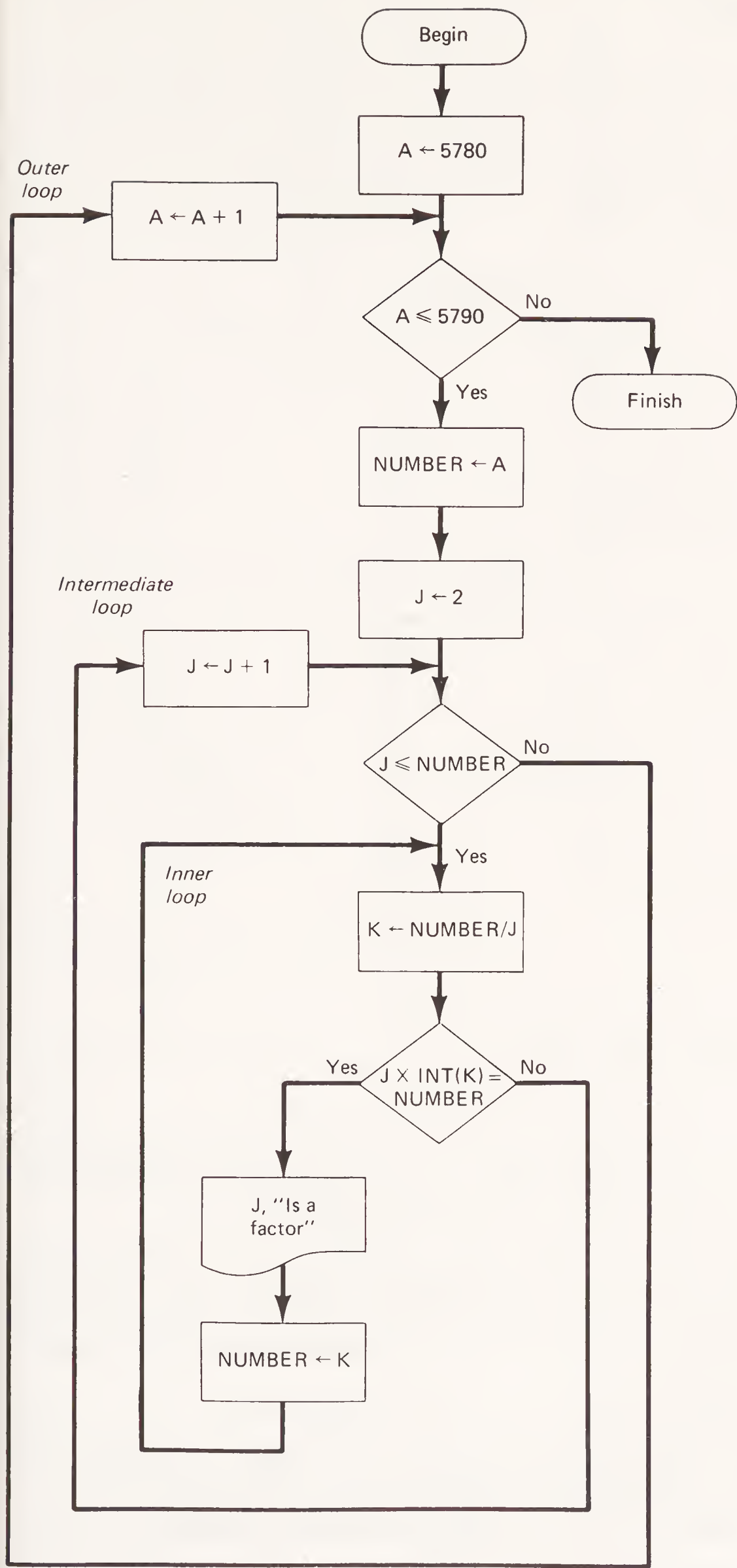


Figure 3-13 Example of three nested loops

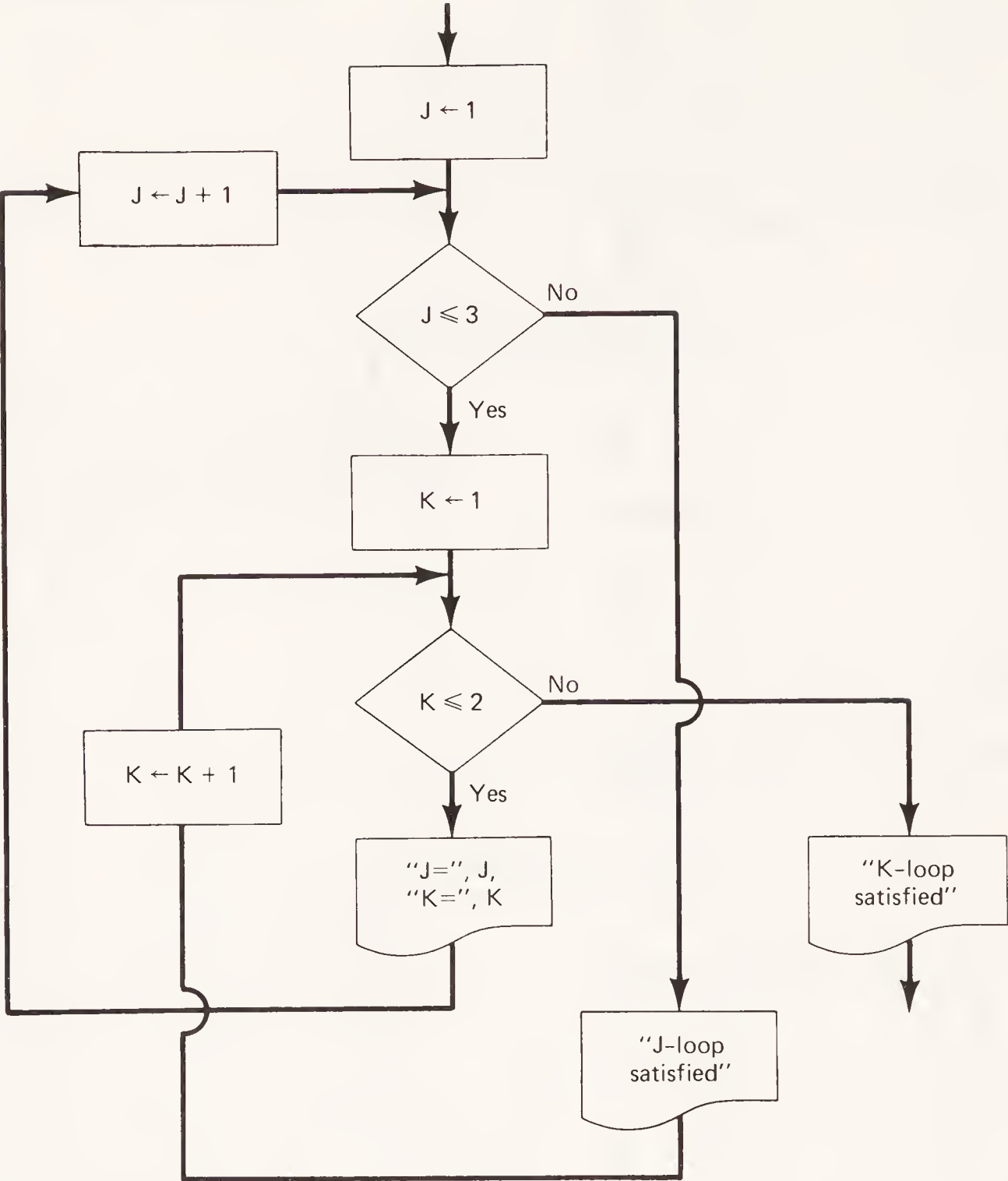


Figure 3-14 Improperly nested loops

messages may arise because the user has not followed the rules of the programming language he is using. These can almost always be corrected with no more effort than a brief consultation with the programming manual.

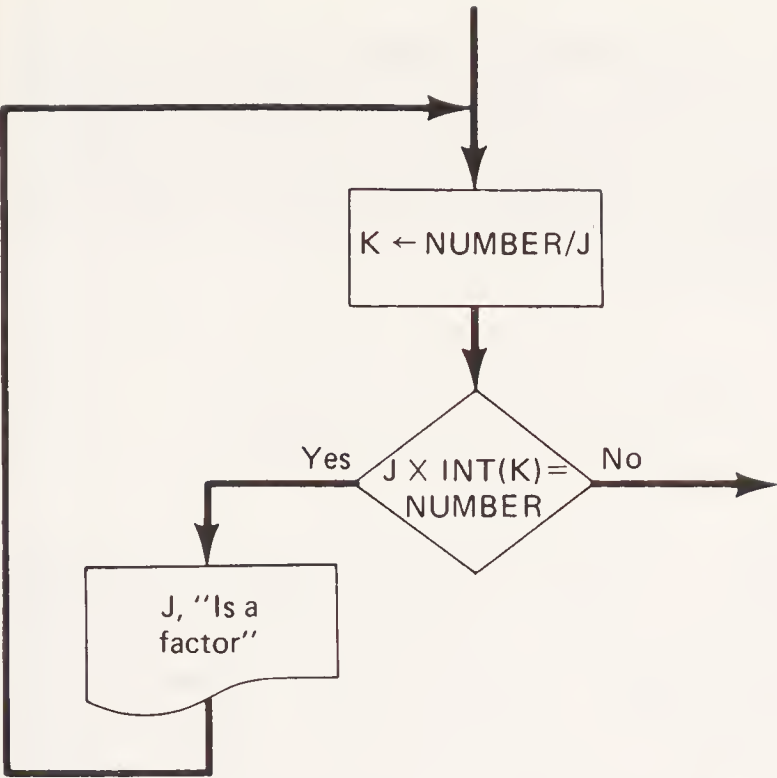
Bugs that produce incorrect answers or no messages at all are frequently more difficult to locate. In these cases, the user has followed all the rules of the programming language; consequently, he usually does not receive any error messages from the computer to guide him in locating his bug. Nevertheless, it must still be corrected.

Using the factoring program presented in the previous section, let's consider the effect of some bugs. As the basis of our discussions, we shall use the flowchart in Figure 3-11 and devote our attention to the inner loop.

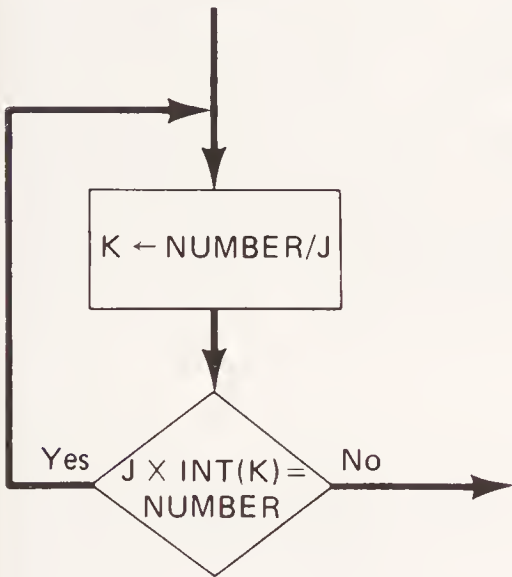
As our first example, suppose we accidentally omit the statement

NUMBER ← K

from the inner loop, as illustrated in Figure 3-15a. What will be the result if the



a. Omission of the assignment statement $\text{NUMBER} \leftarrow K$



b. Omission of PRINT statement

Figure 3-15 Examples of bugs in the inner loop of the program in Figure 3-13

value of **NUMBER** is 10340? The loop begins with **J** equal to 2. As this is a divisor of 10340, **K** will be an integer number. The computer prints

2 IS A FACTOR

and returns to the start of the loop. Since the value of **NUMBER** has not been changed, the computer repeats exactly the same computations on the second pass around the loop. The result

2 IS A FACTOR

is printed and the computations repeated.

In essence, we have constructed a loop for which the normal exit condition will never be satisfied. On each pass around the loop, the output

2 IS A FACTOR

is obtained.

Before considering how to correct this mistake, suppose we first consider what happens when the program is run in this manner. Let's first consider the user of a time-sharing system as described in Section 1.11. Since he is sitting at the terminal watching the results as they are generated, he probably realizes that something is wrong after the computer prints

2 IS A FACTOR

several times. If he does nothing, most time-sharing systems will print

2 IS A FACTOR

all day. The user must initiate the action to *abort* his program, that is, terminate execution. All time-sharing systems provide some mechanism to do this gracefully, that is, other than turning off the power to the terminal. Hopefully, the users will learn how this is to be done *before* they encounter the need for it. For example, on the GE systems, the user need only depress the S key (S for STOP); however, other systems use a different procedure.

In batch-processing systems in which the user runs his own program, similar means for aborting the program are available. But on large batch-processing computers (as described in Section 1.10) where the user submits his job to be processed at a later time, he does not see the answers as they are generated and thus is not in a position to make the decision to abort the job. On such systems, the programmer is generally asked to specify a maximum number of lines or pages of output to be generated. The computer then generates

2 IS A FACTOR

until this maximum is reached. The tendency is for users to grossly overestimate the amount of output expected, which for this case could provide the programmer with a sizable stack of paper with "2 IS A FACTOR" written on each line.

After the programmer has obtained his erroneous results, how does he go about correcting the program? In this case, he may be able to find his error by simply examining the program. The output indicates that he is probably "hung up" in the inner loop. If he cannot detect the problem, the next logical step is to print some intermediate results. Many systems have a "trace" or "debug" feature that automatically prints the result of each arithmetic assignment statement, the route taken from each decision block, etc. Alternatively, the programmer might insert extra PRINT statements in the program. In the case of Figure 3-15a, a PRINT statement could be inserted ahead of the decision block to print the values of NUMBER and K.

Either of the above alternatives would reveal that the value of NUMBER is not changing. From this, the programmer hopefully realizes that a statement is missing.

In the program in Figure 3-15b, the programmer has also omitted the PRINT statement. The computer gets "hung up" in the same loop as before, but no

output is generated. Let's first consider this situation as it applies to a time-sharing user, followed by the same considerations for a batch user.

After waiting at the terminal for a few seconds and getting no output, the time-sharing user suspects that (a) something is wrong with his program or (b) the computer system has ceased to function. As he invariably suspects the latter, most systems provide some means by which the status of the system can be checked. Having verified that the system is functioning, he now aborts his program and begins to look for the bug.

In batch-processing systems, a single user could tie up practically the entire computer for an extended period with a program that is "hung up" in a loop. To avoid this situation, users are required to specify a time limit for each program submitted. If the program has not completed by that time, the program is aborted. Some systems indicate the statement being executed when the abort occurred; others do not. In any case, the user receives a message indicating that an abort was made.

To detect this bug, the programmer will generally examine his program first. He can again turn to the "debug" or "trace" option if necessary. In this case, he should quickly realize that a PRINT statement is needed somewhere in the program.

To verify that a program is processing correctly, many programmers use a "test case" for which the correct solution is known beforehand. Although this approach will often detect incorrect programs, it is not foolproof. If the program yields incorrect answers, then there is certainly a bug somewhere. In some cases, however, a program will produce correct answers for some input but not for others.

As an example of this, suppose we prepare a program to compute the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

The roots are given by the well-known quadratic formula:

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The flowchart of the program to compute the roots is given in Figure 3-16 but contains at least one bug.

To illustrate our point, we have intentionally inserted a bug. Specifically, the parentheses have been omitted from the computation of RE:

$$RE \leftarrow -B/2 \times A \quad \text{instead of} \quad RE \leftarrow -B/(2 \times A)$$

Recall that the left expression is $-ba/2$, not $-b/2a$ as desired.

Suppose we run the program for the following equation:

$$x^2 + 2x + 10 = 0$$

which has the roots $-1 \pm 3i$. If we run the program in Figure 3-16 with $A = 1$, $B = 2$, and $C = 10$, we obtain the results

Real part is -1
Imaginary part is 3

which is the correct answer. This is only because the incorrect expression $-B/2 \times A$ is equivalent to the correct expression $-B/(2 \times A)$ when A equals 1. For any other value of A , different results are obtained.

Similarly, suppose we run the program for the quadratic equation

$$2x^2 + 6x + 4 = 0$$

which has real roots of -1 and -2 . If we run the program in Figure 3-16 for $A = 2$, $B = 6$, and $C = 4$, we obtain the correct answers. In this case, the bug is in a section of the program that is not executed for this set of input data.

In summary, the test cases should be such that all sections of a program are executed at least once. Just because a program gives the correct answers to all test cases, however, does not guarantee that there are no bugs, since, for example, $-B/2 \times A$ equals $-B/(2 \times A)$ for A equal to 1. Running more test cases minimizes the probability that any bugs exist but will never guarantee that none exist.

Another common problem arises when a programmer prepares a program for someone else: that individual may enter data for which the program was never intended to work. For example, what if a zero value of A is entered into the program in Figure 3-16 (after correction of the bug discussed above)? The program terminates on a “division-by-zero” error message because the program as written works only for quadratic equations. When A equals zero, the equation is not quadratic.

3.6. MULTIPLE BRANCHING

Suppose some statistics are available on the number of students in an undergraduate class. Entries are available on each student and contain his or her age and class standing (that is, freshman, sophomore, junior, or senior). The class standing is coded as follows:

1. Freshman
2. Sophomore
3. Junior
4. Senior

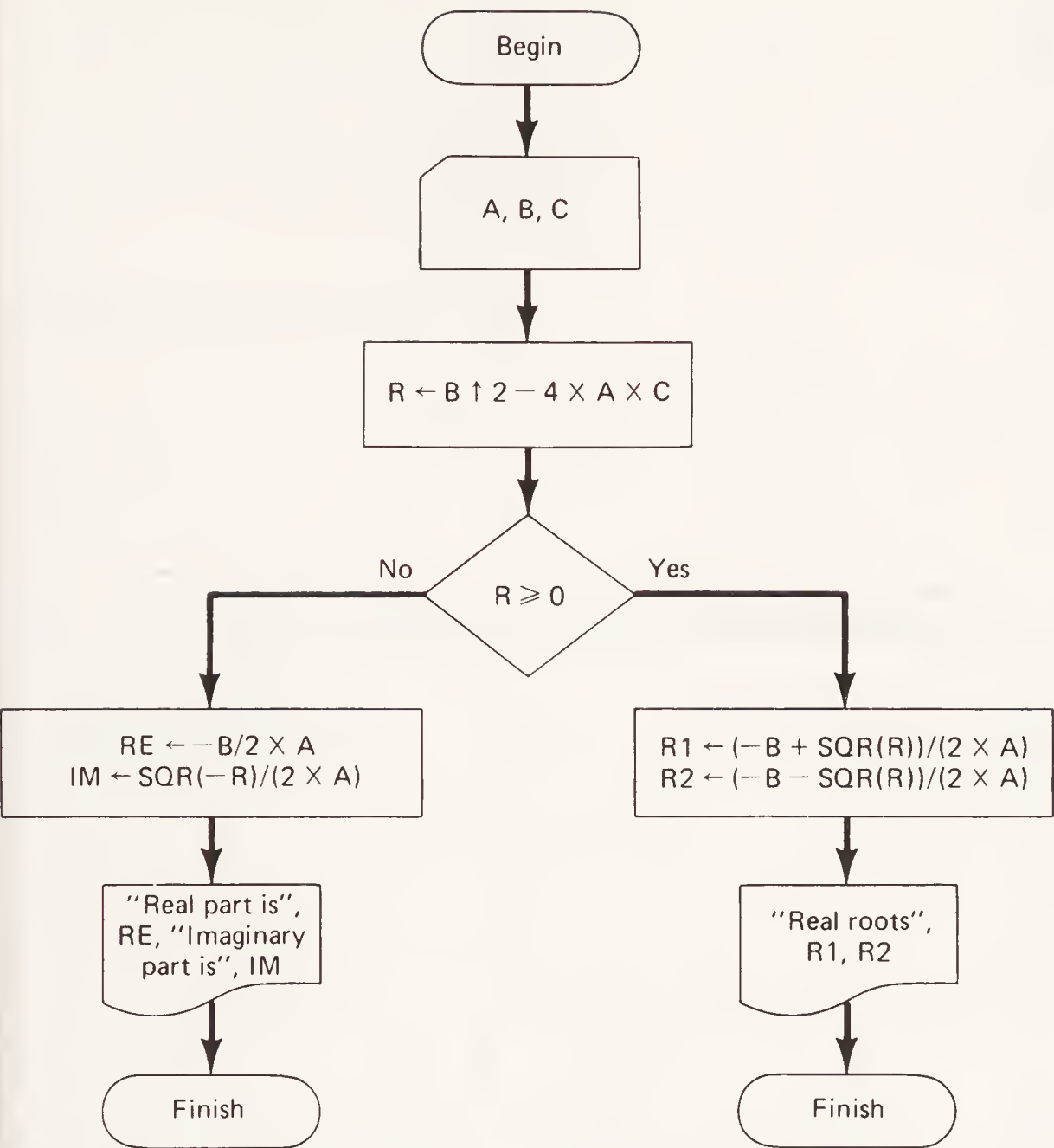
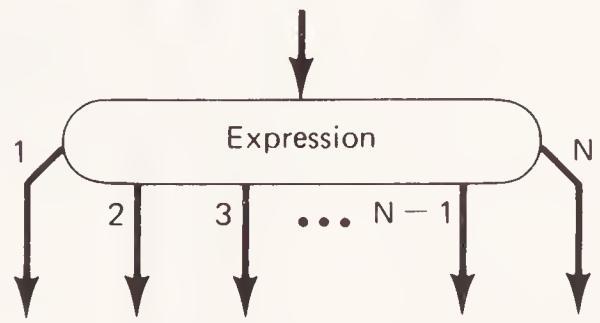


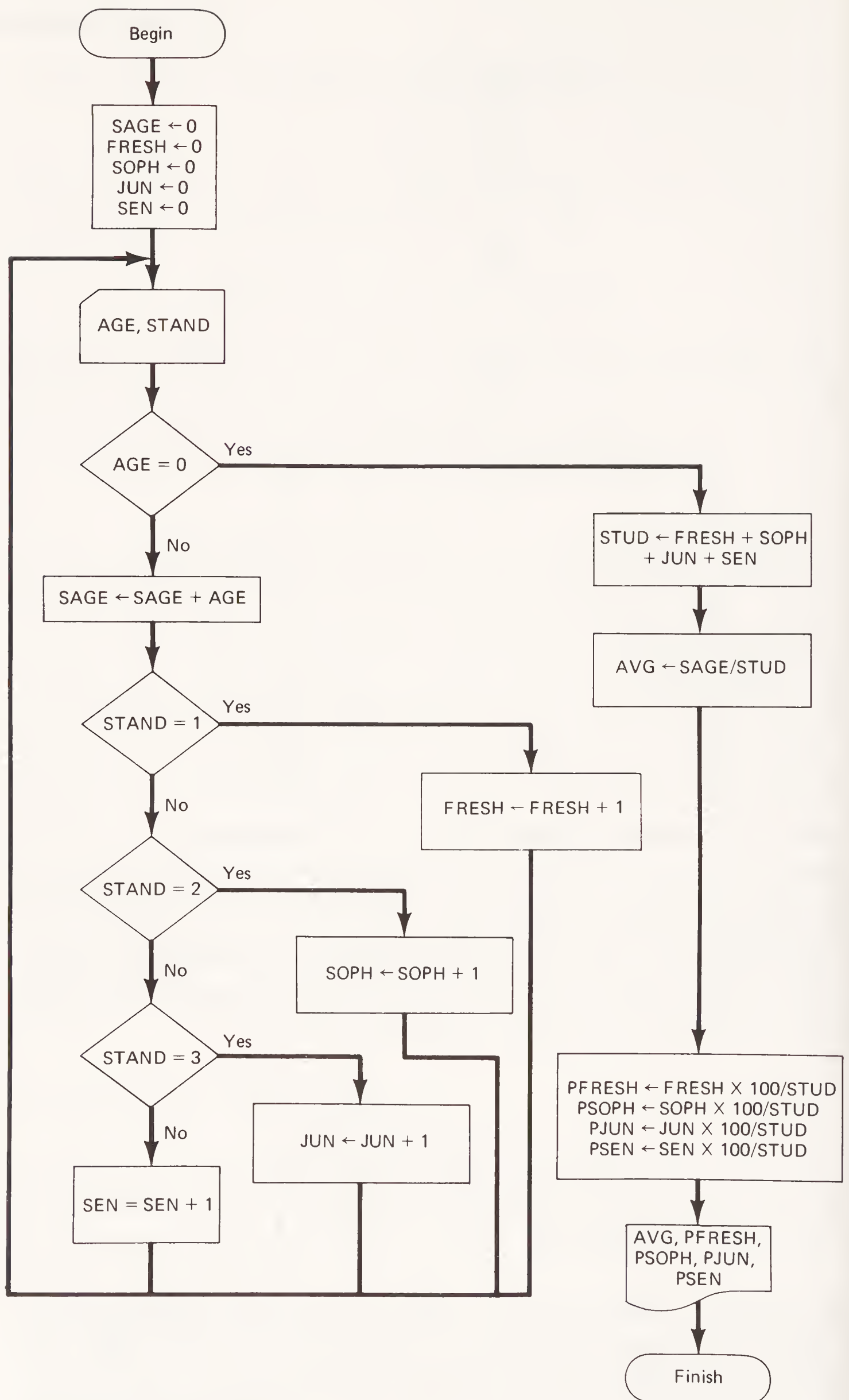
Figure 3-16 Flowchart for computing roots of quadratic equation

Prepare an algorithm to compute the average age and the percentage of freshmen, sophomores, juniors, and seniors.

The flowchart in Figure 3-17a accomplishes this task using three simple-decision blocks. A much more convenient approach is to use a multiple-decision block, which will be represented as follows in flowcharts:

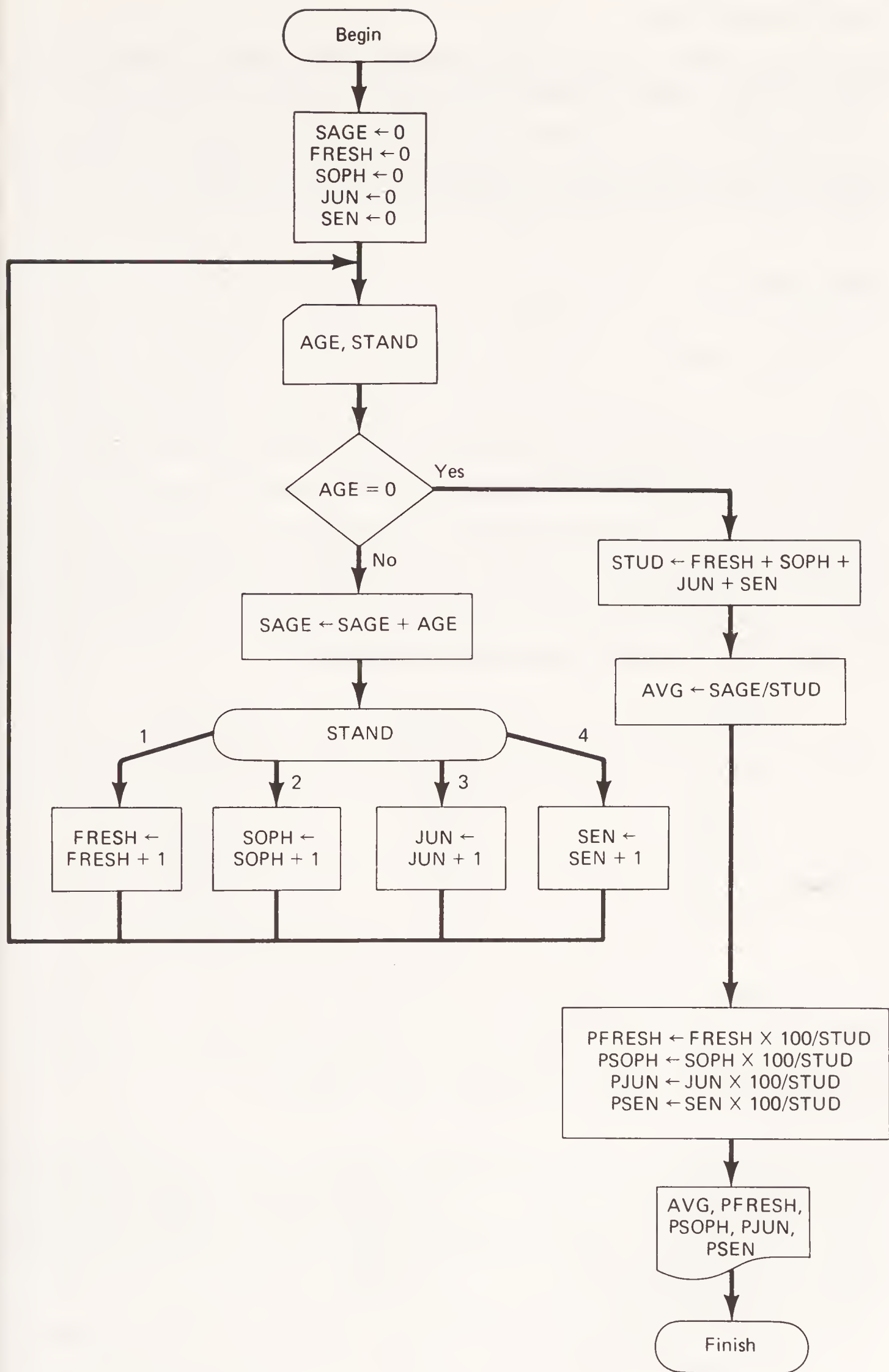


The expression written inside the oval is evaluated and then, if necessary, truncated to an integer. If the result is 1, the branch so marked is taken; if the result is 2, the branch so marked is taken; and so forth up to N. There is no limit on the maximum value of N, but a branch must be provided for every integer



a. Using simple decision blocks

Figure 3-17 Class statistics problem



b. Using multiple branching
Figure 3-17 (Continued)

106
 from 1 to N, inclusive. If the value of the expression (after truncation) is less than one or greater than N, an erroneous condition exists.†

The flowchart of the algorithm utilizing the multiple-branching block for our class statistics problem is given in Figure 3-17b. This flowchart is considerably simpler than the one in Figure 3-17a using simple-decision blocks.

EXERCISES

3-1. Wheeler Dealer is the leading automobile dealer in the state. He now spends \$2,000 a month on advertising and sells 200 cars a month at about \$300 profit per car. From this, he must subtract another \$10,000 as fixed operating costs, which are independent of the volume of sales. An advertising agency tells Wheeler that each time he doubles the amount he spends on advertising, he will increase the volume of sales by 20 percent. Prepare the flowchart for a program that gives Wheeler the amount spent on advertising, the number of sales he makes, and his net profit. Begin with his current status and successively double the amount of advertising until his net profit “goes over the hump,” that is, begins to decline. The output should appear as follows:

Advertising	Units sold	Net profit
2,000	200	48,000
4,000	240	58,000
8,000	288	68,400
16,000	346	77,800
32,000	415	82,500
64,000	498	75,400

3-2. In preparing the flowchart for Exercise 2-22, the most convenient approach is to use the INT function. An alternate approach is to use loops to determine how many of each type of coin is to be given in change. Prepare such a flowchart, thereby avoiding the use of the INT function.

3-3. A man borrows \$200 at an interest rate of 1 1/2 percent per month. He proposes to repay this at the rate of \$10 per month. How many full payments must he make, and what must be paid in the final month to leave a balance of exactly zero? Prepare a flowchart for this program. The calculations for each month should be as follows:

- Calculate the interest for that month, which is 1 1/2 percent of the current balance.
- Add the interest to the current balance to obtain the total amount owed at the end of the month (before receipt of payment).

†In some programming languages, this does not constitute an error condition. Some prespecified action will be taken if the value of the expression is outside these limits.

- c. Subtract the payment to obtain the new balance. For the first month, the interest is $(1\frac{1}{2}\text{ percent}) \cdot (\$200) = \$3$. The new balance is $\$200 + \$3 - \$10 = \$193$.

3-4. A man plans to invest \$500 per year at 6 percent interest. How many years will be required to accumulate \$10,000? Prepare the flowchart for a program to compute this. Print the number of years along with the total value of his investments when they first exceed \$10,000. The calculations proceed as follows. After the first year, the value of the investment is $\$500 \cdot (1 + 0.06) = \$530$. After the second year, the value is $(\$500 + \$530) \cdot (1 + 0.06) = \$1,091.30$.

3-5. Consider writing a program that would compute the arithmetic average of the absolute values of a set of data. Suppose, however, that it is undesirable to count the number of values in this set of data. If it is known beforehand that none of the values will be exactly zero, the computer can be programmed to read successive values until a value of zero is detected as discussed in the chapter. Prepare the flowchart for a program that computes the arithmetic average of the values 0.7, 1.9, -2.2, 7.1, 17.0, -4.2, 7.9, and -5.1. The program should print and appropriately label the total number of values and the arithmetic average.

3-6. Prepare the flowchart for a program similar to the one in the previous exercise except that the geometric mean of the absolute values is computed. The geometric mean of n values is the n th root of their product.

3-7. To seven decimal places, the fraction $1/7$ is given as 0.1428571. Suppose you wanted it expressed to 20 decimal places. Can you devise a procedure to calculate the digits one by one?

3-8. One method of finding the square root is to use the iterative formula

$$S_{i+1} = \frac{1}{2} \left(\frac{X}{S_i} + S_i \right)$$

where S_i is the estimate of X on the i th iteration. For $X = 300$ and starting with $S = 15$, continue to iterate until the value of S does not change in the fourth decimal place between successive iterations. As output, print the results of this procedure along with the results using the SQR function. What changes would you make in the program to compute the square root of 10,000,000? of 0.001? (*Hint*: One way to make the test for changes in the fourth decimal is to express the equation in the form

$$S_{i+1} = S_i + \frac{1}{2} \left(\frac{X}{S_i} - S_i \right)$$

Now when $1/2 | X/S_i - S_i |$ is less than 0.0001, the solution is found according to our criterion.)

3-9. Another way to formulate the method in the above exercise is as follows. To calculate the square root of a number, say b , select any number, say g_1 , and compute another number, say h_1 , as follows:

$$h_1 = \frac{b}{g_1}$$

The next step is to average g_1 and h_1 to obtain g_2 :

$$g_2 = \frac{g_1 + h_1}{2}$$

Then h_2 is computed as before:

$$h_2 = \frac{b}{g_2}$$

This procedure can be generalized as follows:

$$h_i = \frac{b}{g_i}$$

$$g_{i+1} = \frac{g_i + h_i}{2}$$

The sequences $g_1, g_2, \dots, g_i$ and $h_1, h_2, \dots, h_i$ both approach the square root of b . The stopping criterion is whenever

$$|g_i - h_i| < \epsilon$$

where ϵ is a small but positive number. Use this program to take the square roots of 4 and 2, starting with $g_1 = 1$ in both cases and using $\epsilon = 0.00001$.

3-10. The cube root S of a number X can be calculated using the iteration formula

$$S_{i+1} = \frac{2}{3}S_i + \frac{X}{3S_i^2}$$

The iterative procedure may be initialized by setting $S_1 = 1$. Prepare the flowchart for a computer program to take cube roots of numbers in this manner. For informative purposes, print the value of S after each iteration. Terminate the procedure when the change in S from one iteration to the next is less than 0.01 percent of the current value of S . The above formula will work for positive or

negative values of X , so your program should work likewise. Run the program for $X = 235$ and $X = -91.6$.

3-11. Let the function $F(X)$ be defined as $(X + 2)/(X + 1)$. Define the sequence

$$a_i = F(a_{i-1})$$

Prepare the flowchart for a computer program to compute the first ten terms of the sequence.

3-12. The two sequences

$$a_i = \sqrt{a_{i-1} b_{i-1}}$$

$$b_i = \frac{a_{i-1} + b_{i-1}}{2}$$

$$a_1 = a, b_1 = b, a \leq b$$

converge to the same limit, called the “arithmetic-geometric mean.” Prepare the flowchart for a program that reads a and b , checks to be sure that $a \leq b$ (if not, interchange a and b), and computes the sequence until $|a_i - b_i| < \epsilon$. Run the program for $a = 2$, $b = 7$, and $\epsilon = 0.00001$.

3-13. The function e^x can be represented by the infinite series

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

Note that the n th term is simply the previous term multiplied by $x/(n - 1)$. Compute e^x using this series, stopping when the magnitude of the term being added is less than 10^{-6} . Run the program for $x = 1.7$ and $x = -1.8$.

3-14. A plant whose production rate is P lb/year produces a product whose value is V \$/lb. The product costs C \$/lb to produce. The gross profit is then $P \cdot (V - C)$. The total tax rate is 52 percent, so that the profit after taxes is 0.48 times the gross profit. If the plant costs B \$ to build, the payout time in years is B divided by the profit after taxes.

Suppose the values of C , B , and P are known to be \$1.50/lb, \$900,000, and 200,000 lb/yr, respectively. Prepare the flowchart for a program to calculate the payout time for values of V from \$2.75/lb to \$3.25/lb in increments of \$0.05/lb. The listing should be in columnar fashion, with a value of V and a corresponding value of the payout time.

3-15. A man invests \$600 per year at 8 percent interest. How much will he have after ten years?

Prepare the flowchart for a program to calculate this using a loop. Read the amount invested annually, the interest rate, and how many years from now that the value of his investment is to be calculated.

3-16. At an interest rate of 6 percent, how much must you deposit at the first of each year so that at the end of five years you will have \$8,000?

Prepare the flowchart for a program to calculate this. Assume \$1.00 is deposited each year and calculate the amount available at the end of five years as per the previous problem. The amount to be deposited is \$8,000 divided by this value. Read the interest rate, how much is to be accrued, and how many years deposits will be made.

3-17. One calculation frequently encountered in financial analyses is the computation of depreciation. In addition to the sum of the years' digit method presented in the chapter, the double-declining-balance method is also popular. In this method, the depreciation for the year is $2/n$ (n = number of years over which item is depreciated) times the book value at the beginning of the year. Prepare the flowchart for a program that reads the amount to be depreciated and the number of years over which the depreciation is to be made. The output should be tabulated as follows for depreciating, \$30,000 over a 5-year period:

Year	Beginning book value	Depreciation	Book value at end of year
1	30,000	12,000	18,000
2	18,000	7,200	10,800
3	10,800	4,320	6,480
4	6,480	2,592	3,888
5	3,888	1,555	2,333

3-18. Prepare the flowchart for a program to read N and calculate $N!$. N is always a nonnegative number, but may be zero ($0! = 1$). The output should be N and $N!$ [$N! = 1 \times 2 \times \cdots \times (N - 1) \times N$]. $N!$ is pronounced " N factorial."

3-19. In business applications, a quantity known as the *capital-recovery factor* is defined as follows:

$$\text{Capital-recovery factor} = \frac{i(1 + i)^n}{(1 + i)^n - 1}$$

where i is the interest rate and n is the number of years. Prepare the flowchart for a program to print a tabulated set of values for $n = 1$ through $n = 25$ for $i = 8$ percent. The output should appear as follows:

1	1.08000
2	0.56077
3	0.38804
	etc.

3-20. A man borrows \$100 at an interest rate of 1 1/2 percent per month. If he pays \$10 at the end of each month, how much does he owe at the end of ten months? Prepare the flowchart for a program to solve this problem. Read the amount he borrows and the amount he pays each month. To be sure we are in agreement as to how the calculations are to be done, note that the interest for the first month is $0.015 \times \$100 = \$1.50$. Since he pays \$10 at the end of the month, he owes $\$100 + \$1.50 - \$10 = \$91.50$. Use a loop to iterate these calculations for each month.

3-21. On the interval $0 < X \leq 2$, $\log (X)$ can be represented by the following infinite series:

$$\log (X)=\frac{\left[(X-1)-\frac{(X-1)^2}{2}+\frac{(X-1)^3}{3}-\dots\right]}{2.302585}=\frac{\sum_{n=1}^{\infty} \frac{(-1)^{n-1}(X-1)^n}{n}}{2.302585}$$

Prepare the flowchart for a program that reads a value for X and sums the series, printing the answer after 3, 5, 10, 50, and 100 terms. Also, print the answer using the library logarithm function for comparison. Run the program for $X = 1.8$. In programming assume X will always be in the above interval.

3-22. The binomial coefficients $\binom{n}{j}$ are given by the following expression:

$$\binom{n}{j}=\frac{n!}{j!(n-j)!} \qquad j=0,1,\dots,n$$

The Binomial Coefficients for N = 8 are

N	J	Coefficient
8	0	1
8	1	8
8	2	28
8	3	56
8	4	70
8	5	56
8	6	28
8	7	8
8	8	1

Prepare the flowchart for a program to read a value for n (use $n = 8$ for this case), compute the binomial coefficients, and print the results as in the illustration. Assume n is always greater than 1. The cautious programmer will note that the first term is 1, and each succeeding term is simply $(n-j+1)/j, j = 1, 2, \dots, n$, times the preceding term. Programming in this manner avoids extremely large numbers which may occur if the factorials are evaluated directly. (This is not the case for $n = 8$, and such programming may be used if desired.)

3-23. Suppose at time zero a worm is 2 feet away from an apple. At time zero, he moves one half the distance between him and the apple; that is, he moves one

foot. Ten minutes later, he moves one-half the remaining distance, or one-half foot. He continues this at regular intervals of 10 minutes, always moving only one-half of the distance between him and the apple at each step. How long does it take him to get to within 0.1 inch of the apple?

3-24. Exercise 2-7 gave a formula for computing the sum of an arithmetic progression. Prepare a flowchart using a loop to sum the series term by term and compare with the answer given by the formula in Exercise 2-7.

3-25. Exercise 2-8 gave a formula for computing the sum and last term in a geometric progression. Prepare a flowchart using a loop to sum the series term by term and to compute the last term. Compare the results with those given by the formulas in Exercise 2-8.

3-26. Exercise 2-9 gave a formula for computing the sum of a combined arithmetic and geometric progression. Prepare a flowchart for a program to sum these series term by term and compare with the answer given by the formula for Exercise 2-9.

3-27. Exercise 2-17 described the problem of updating a student's grade-point average. Using a loop, program this exercise as follows:

1. Read the student's identification number, his old grade-point average, and the number of hours to which his old grade-point average applied.
2. Enter the credit hours for one course and the numerical equivalent for the grade, make the necessary updates, and then proceed to the values for the next course.
3. An entry of zero credit hours signifies end-of-data.

3-28. Make the modification suggested in the previous exercise to Exercise 2-29 also.

3-29. Prepare the flowchart for a program to evaluate a polynomial such as the following:

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

The computations are to proceed as follows:

1. Read a value for x .
2. Read a value for a_0 and assign to the initial value of the function.
3. Read a_1 , multiply by x , and add to the value of the function.
4. Repeat the previous step for successive values of the coefficients, always multiplying by the appropriate power of x and adding to the value of the function.
5. Entry of a coefficient whose value is greater than 10^{25} will signify that the previous value entered was the final coefficient.
6. The program should print the value of $f(x)$.

Be sure to use a loop.

3-30. Exercise 2-36 presented a formula for applying discounts to identical items in an order. Let's extend to the situation where several items may appear in a single order, and the formula is to be applied to each. For example, consider the following order:

Item no.	Units	Unit price
407	42	\$ 4.45
219	7	27.15
375	125	6.25
129	19	13.42

For each item, the program should read the item number, the units ordered, and the unit price. The output should consist of the input followed by the total cost for all units of a given item and the average price. Finally, the program should print the total cost of all items in the order. Signify end-of-data by entry of a zero item number.

3-31. In Appendix E, a technique is described for converting an integer decimal number to a binary number. Prepare a flowchart for a program to make this conversion, printing the digits one by one as they are generated. Read the decimal number.

3-32. Modify the previous exercise so that the number is converted to any base other than 10. Read the decimal number and the base of the numbering system to which it is to be converted.

3-33. In Appendix E, a technique is described for converting a decimal fraction to a binary fraction. Prepare a flowchart for a program to make this conversion, printing the digits one by one as they are generated. Read the decimal number and the number of digits for the binary fraction.

3-34. Modify the previous exercise so that the fraction is converted to any base other than 10. Read the decimal fraction, the base of the numbering system to which it is to be converted, and the number of digits to appear in the resulting number.

3-35. Use the multiple branching feature to program Exercise 2-33.

3-36. Suppose we should like to compute the sum of squares of the integer numbers. For the first five integer numbers, the computation is

$$1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$$

The question is whether or not the result of such a computation is ever a perfect square itself. The answer to this question is yes, since a series of only one term sums to 1, which is a perfect square. But are there any others? Prepare the flow-chart for a program to solve this problem, stopping after 200 integers if no perfect square has been found.

3-37. The number 48 has the following interesting characteristics:

1. Addition of 1 produces an exact square ($49 = 7^2$)
2. Addition of 1 to its half produces an exact square ($25 = 5^2$).

Write a computer program that locates the next higher number with this interesting property.

4

SUBSCRIPTED VARIABLES

In previous chapters, a single storage location was reserved for each variable. Therefore, for each variable, we could store a single numerical value in this storage location. In certain types of problems, use of a unique name for each item of information is extremely inconvenient. By reserving several storage locations for certain variables, the subscripted variable feature permits the same name to be associated with several numerical values. A subscript then designates the specific storage location or value from all of those associated with the variable.

If the same operation is to be performed on the quantity stored in each of the storage locations, an iteration counter can be used to construct a loop around a group of statements that designate the desired operation. The statements will contain one or more subscripted variables, the value of the subscript being the same as or at least directly related to the index of the iteration counter. The combination of the iteration counter and subscripted variables is the most powerful feature of most programming languages. However, subscripted variables can be used without using loops, and vice versa.

Variables for which only one storage location is reserved will be called *simple* variables. The term *array* will be used to denote variables for which more than one storage location are reserved.

4.1. THE NEED FOR SUBSCRIPTED VARIABLES

In Section 3.1, we examined two approaches to computing the average age of a class of seven students. As output from this program, suppose we want to print the student's age and the difference between his age and the average age.

While we could use individual simple variables for the age of each student, this approach suffers the same disadvantages as outlined in Section 3.1. The use of a loop to compute the average age was shown to be much more convenient

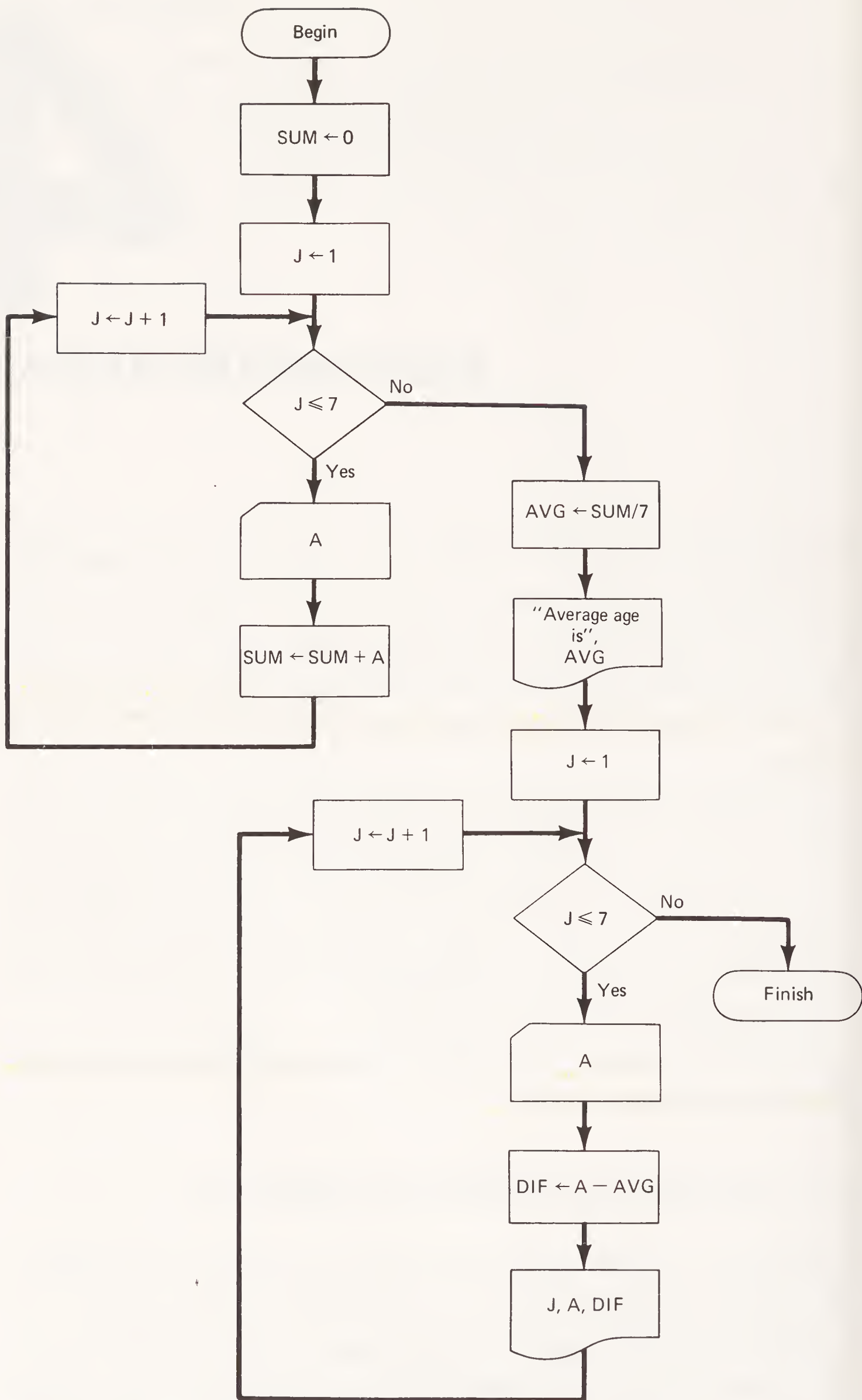


Figure 4-1 Re-entry of data

and essentially consisted of the upper section of the flowchart in Figure 4-1. In this case, the age of a student is entered by means of simple variable A, which is then added to a running total. Since the current student's age is not needed on the next pass around the loop, the variable A can be reused.

While this approach is entirely adequate when we compute the average, problems are encountered when we try to compute the difference of the student's age from the average. Since the difference from the average cannot be computed until after the average age has been calculated, the top loop cannot compute the differences. This must be done in a subsequent section of the program. But upon exit from the upper loop in Figure 4-1, variable A contains the age of the seventh student, that is, the last value entered. The ages of the first six have been lost.

One possible approach is to re-read the data, as illustrated in Figure 4-1. After the average age has been computed, a second loop is entered. Upon each iteration, a student's age is read, the difference from the average computed, and the results printed. While re-entry of data is certainly feasible, a more desirable alternative for most problems is the use of subscripted variables. In general, re-entering of data should be avoided if at all possible.

Use of subscripted variables avoids this difficulty in all but very large data sets. To illustrate the concept of subscripted variables, consider the following set of data, which we might refer to as a *list* of the student's ages:

Student number	Age	Subscripted variable
1	20	A ₁
2	31	A ₂
3	18	A ₃
4	22	A ₄
5	21	A ₅
6	22	A ₆
7	27	A ₇

The variable A denotes the age of all the students. To designate individual students, a subscript is attached.

In this example, the variable A is associated with the list of students' ages. More elegantly, the list is said to be a *single-subscripted* array, namely, 20, 31, 18, 22, 21, 22, 27. The subscript simply denotes the location within the array; that is, A₄ is the fourth element in the array, namely, 22.

Figure 4-2 gives the flowchart of our problem utilizing subscripted variables. Note the use of the iteration counter to vary the subscript J from 1 to 7 as the values are read. In this manner, each age is stored in a unique storage location. All ages are therefore available after exiting from the upper loop. In the lower loop, the iteration counter again varies the subscript J as the differences from average are computed. The variable DIF, however, is not subscripted. Since DIF

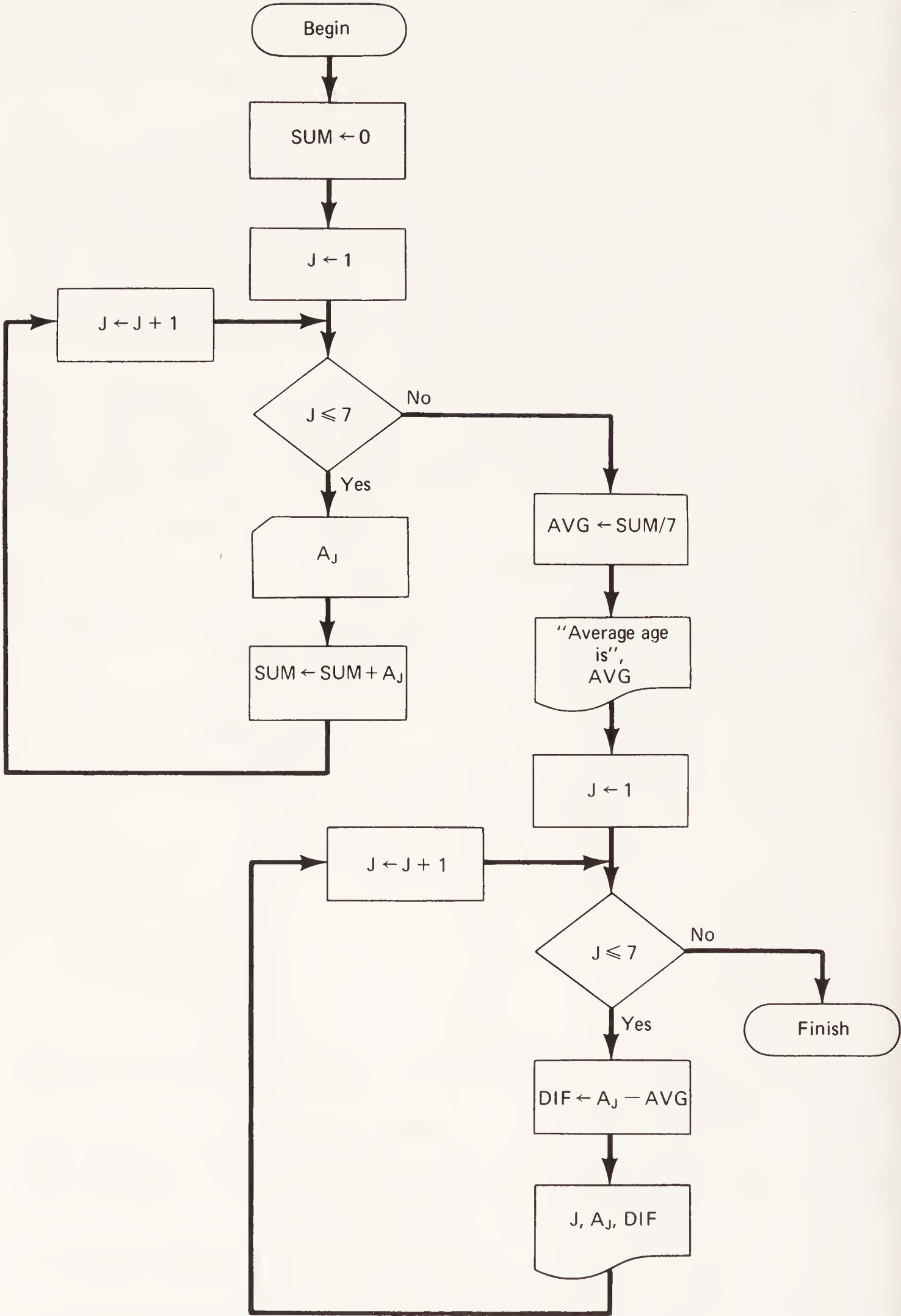


Figure 4-2 Use of subscripted variables

is computed and immediately printed, there is no need to preserve these values for later use in the program. Thus, a simple variable is adequate and should be used instead of a subscripted variable. In a later section, we shall consider the question of when to use subscripted variables and not simple variables.

If the class of students was extremely large, there may not be enough storage locations in the computer to store the entire array A. In this case, the alternative in Figure 4-1 must be used.

4.2. MEMORY ALLOCATION

In Chapter 1, we noted that the processing of a program required two distinct phases—a compilation phase and an execution phase. One important aspect of the compilation phase is the assignment of variable storage.

As the first example, let’s allocate storage for the variables that appear in the flowchart in Figure 4-1. We simply scan through the flowchart, examining the blocks in sequence for the first appearance of a variable or constant. If the first storage location reserved for a variable is location 1000, storage would be allocated as shown in Table 4-1a. A total of eight storage locations are required for all variables and constants.

Next, let’s allocate storage for the variables appearing in the flowchart in Figure 4-2. Again, starting with location 1000, storage is arranged as in Table 4-1b. A total of fourteen storage locations are required in this case, as compared with eight as before.

The array is generally stored in consecutive storage locations in memory. A location is reserved for each integer value of the subscript between 1 and the maximum value. Even if one of the subscripts, say 2, were never used in the program, a location would still be reserved by most systems. In fact, to store and retrieve values in an array, the only item required is the address of the location

Table 4-1. Allocation of Memory

A. FOR FIGURE 4-1		B. FOR FIGURE 4-2	
Location	Quantity	Location	Quantity
1000	SUM	1000	SUM
1001	0	1001	0
1002	J	1002	J
1003	1	1003	1
1004	7	1004	7
1005	A	1005	A ₁
1006	AVG	1006	A ₂
1007	DIF	1007	A ₃
		1008	A ₄
		1009	A ₅
		1010	A ₆
		1011	A ₇
		1012	AVG
		1013	DIF

used for the first element. For array A in Table 4-1b, A_1 is stored in location 1005. If we take the address of the first storage location, namely, 1005, add the value of the subscript, and subtract 1, the address of the corresponding storage location is obtained. That is, to determine the address of the storage location for A_6 , simply add 5 to 1005 to obtain 1010, which is the correct address, as can be verified from Table 4-1b.

Some compilers reserve a storage location for subscript zero, while others do not. On this point, the programmer must consult manuals specific to his system to ascertain if zero subscripts are allowed. A few systems even permit negative subscripts.

During the execution phase, some systems check for subscripts out of range, while others do not. For example, suppose by mistake the programmer requested that the value of A_8 be printed in the program in Figure 4-2. Since only seven locations were reserved (see Table 4-1b), some systems would generate an error message. Others do not check. They simply add the value of the subscript (8 in this case) to the address of A_1 (in this case, 1005) and then subtract 1 (to give 1012 for this example) to obtain the address of the quantity to be printed. For this example, the contents of address 1012 are printed, which is really the value of variable AVG.

In the flowchart in Figure 4-2, the iteration counters clearly indicate that seven storage locations are needed for array A . Unfortunately, current compilers are not “smart” enough to make such judgments. Even if they could, a more general approach to programming this problem would be to read the number of students from a data card. Recall that no cards are read until the execution phase, whereas storage is being allocated in the compilation phase. This value is not available to the compiler, which then has no way to determine how many storage locations are needed.

Many current compilers require that the programmer specify somewhere in his program the number of locations to be reserved for each array. In effect, this specifies the maximum subscript value to be used in the program. At least one system, namely, Basic, has a “default” option by which it reserves a preassigned number of storage locations (typically ten) for any array whose size is not specified.

The discussion in this section actually applies only when the compiler reserves the storage locations for variables. An alternative to this approach is to use *dynamic storage allocation*, whereby storage locations are reserved for variables only as they are used during the execution phase. This approach differs from the above discussion in several respects:

1. The maximum size of the array need not be specified to the compiler.
2. The elements of the array are not necessarily stored in consecutive storage locations.
3. If one element, for example, A_2 in the array of ages discussed in this section, were not used, no storage location would be reserved for it.

While dynamic storage allocation offers several advantages, its implementation is more difficult, and therefore few systems have incorporated it.

4.3. USE OF SUBSCRIPTED VARIABLES

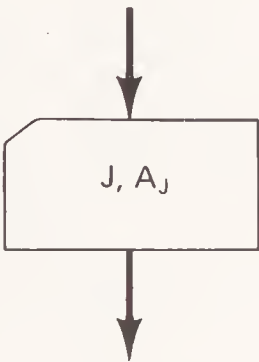
Before illustrating the use of subscripted variables in some examples, a few points regarding their use should be made.

- 1. *The subscript is not part of the name of the array.* That is, A_J and A_K both refer to array A , the subscript value denoting only the location within the array. That is, A_J refers to the J position in array A ; A_K refers to the K position in array A .
- 2. *Expressions may be used as subscripts.* That is A_{J+1} or $A_{K/2+1}$ are acceptable. Although some programming languages have restrictions on admissible forms of expressions in subscripts, we shall make none.
- 3. *All variables appearing in expressions used as a subscript must be defined prior to their use.* That is, in the statement

$$A_{2 \times K + 1} = 4.708$$

a value must have been assigned to K previously in the program. If K were assigned a value of 4, then 4.708 would be stored in the ninth position of array A .

In READ statements, we shall consider the input to occur in the sequence specified. Therefore, the block



is accepted, since J is entered first, followed by the value to be stored in element J of array A .

- 4. *Storage locations are reserved only for integer subscript values.* Some systems will only accept integer values of the subscript, while others either round or truncate the fractional parts.

As the first example of the use of subscripted variables, suppose we consider a program to update an inventory for items received. The number of items in inventory prior to receipt of a shipment is available. For each stock number, the

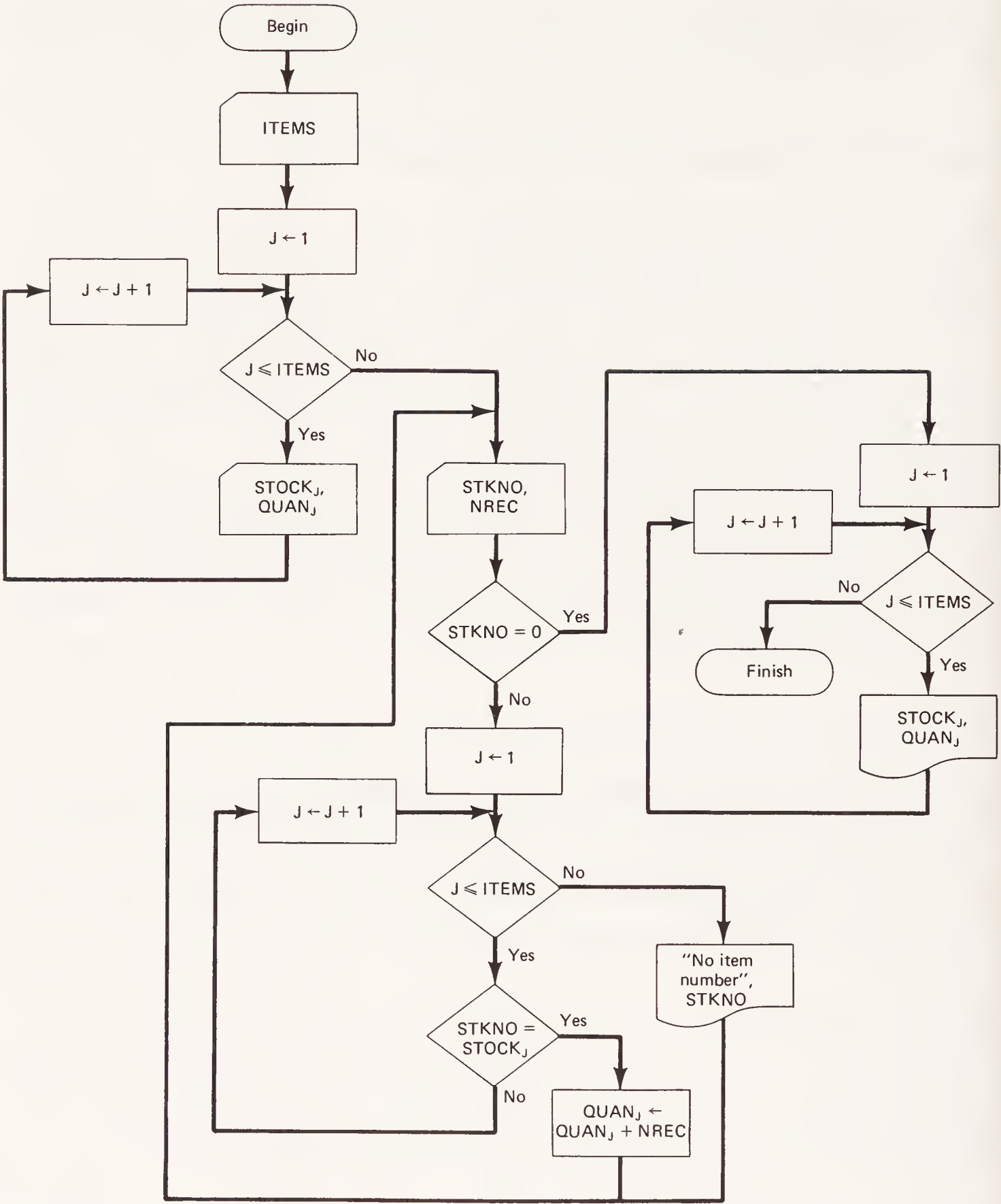


Figure 4-3 Inventory program

number of corresponding items in inventory is available. We also have the stock number and quantity of each item received. In their current status, the stock numbers of the items in inventory are not ordered in any way; that is, they are not in ascending or descending order. Nor are the stock numbers of items received ordered in any way. In fact, either of the following conditions may hold:

1. A shipment does not include all items in inventory; that is, the stock-received data does not contain an entry for each item in inventory.

- 2. In some shipments, two or more entries may exist for some items in inventory.
- 3. Owing to errors in handling, occasionally an entry is encountered which does not correspond to a stock number in inventory.

In the latter case, an appropriate message should be printed.
As the final step, the updated inventory should be printed.

The flowchart for this program is given in Figure 4-3. The total number of items in inventory (ITEMS) is entered, followed by the stock number, STOCK_J, and corresponding quantity, QUAN_J, for each item in inventory. An iteration counter is used to read each entry and store it in the appropriate location in arrays STOCK and QUAN.

After reading the current inventory, the program then begins to process the items-received entries. An entry consisting of a stock number, STKNO, and the number of items received, NREC, is read. The stock number, STKNO, is then compared with each entry in array STOCK. When an element is located in array STOCK that equals the stock number of the item received, STKNO, the corresponding element in array QUAN is increased by the number of items received, NREC. If, after comparing STKNO with each element in array STOCK, no equality condition was found, the program then prints the message

No item number

followed by the value of STKNO.

In either case, the program returns to the READ statement to process another entry. Upon encounter of the “end-of-data,” the program then prints the updated inventory.

Arranging Elements in Ascending Order

Since the stock numbers in the inventory considered in the previous example were in random order, this would make it quite difficult for a clerk to look up in the output list and determine how many items of a certain type are in stock. If the output were in either ascending or descending order in regard to the stock numbers, this would be much easier.

Let’s first consider how to arrange an array of elements in ascending order. In this section we shall illustrate a procedure known as the *replacement sort*; in the next section, we shall illustrate a procedure known as the *exchange sort*. To illustrate the replacement sort procedure, we shall consider the following array:

Element	Value
1	1407
2	2132
3	1107
4	5072
5	1021

124 One approach to arranging elements in ascending order is to first locate the smallest element. In the above array, the smallest is element 5. We can then interchange element 1 with element 5 to obtain

Element	Value
1	1021
2	2132
3	1107
4	5072
5	1407

The next step is to repeat this procedure on elements 2 through 5. The smallest of these is element 3. Interchanging with element 2 gives

Element	Value
1	1021
2	1107
3	2132
4	5072
5	1407

Next, we repeat this procedure for elements 3 through 5. The smallest of these is element 5. Interchanging with element 3 gives

Element	Value
1	1021
2	1107
3	1407
4	5072
5	2132

Finally, we repeat this procedure for elements 4 and 5. Since element 5 is the smallest, we interchange with element 4 to obtain

Element	Value
1	1021
2	1107
3	1407
4	2132
5	5072

The array is now in ascending order.
The flowchart in Figure 4-4 outlines this approach for an array A containing

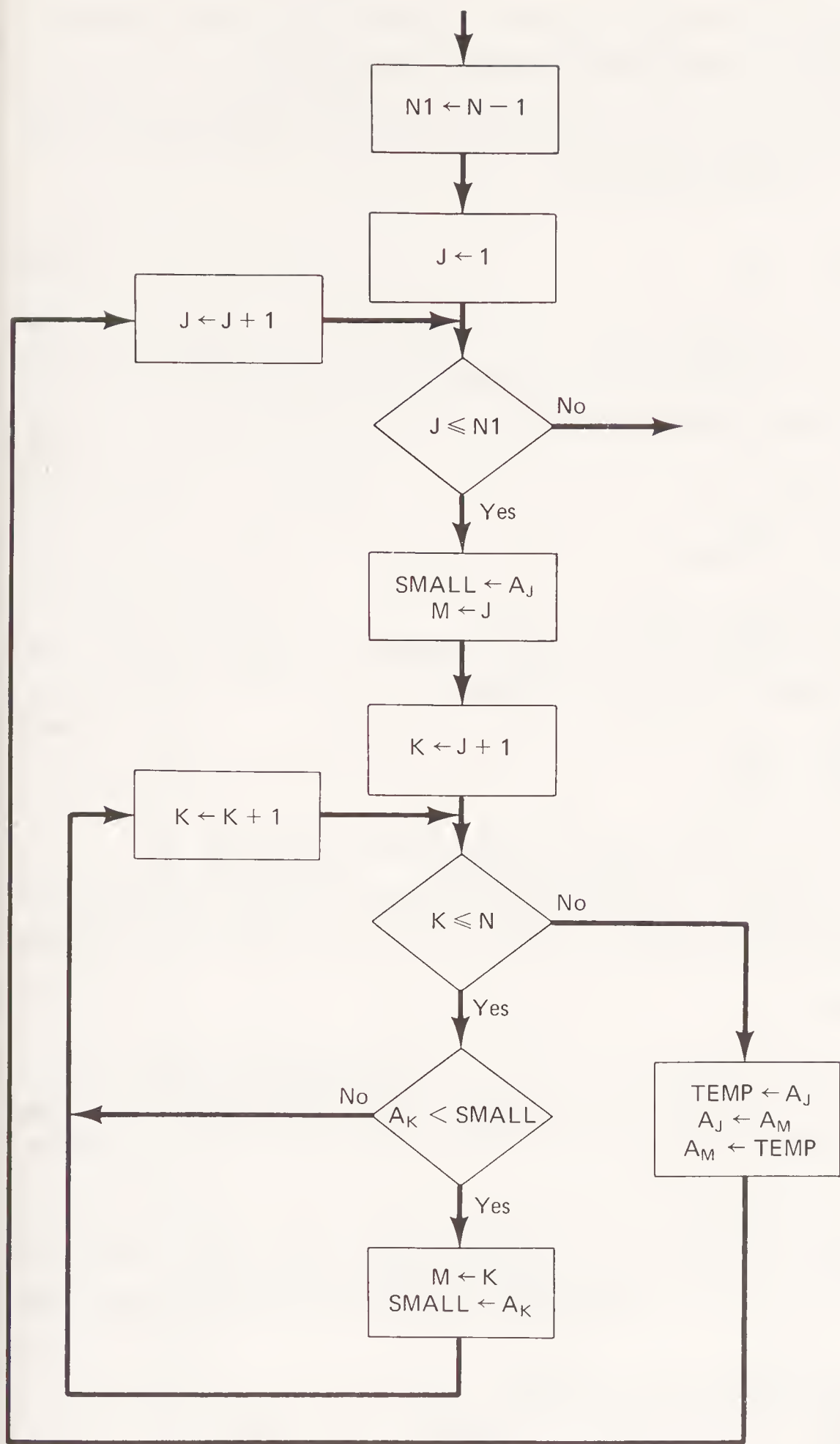


Figure 4-4 Flowchart for arranging elements of an array in ascending order

N elements. In locating the smallest of elements J through N , element J is assumed to be the smallest. Its value is stored in variable $SMALL$ and its subscript in variable M . By means of a loop, each element is compared with $SMALL$ until a smaller one is found. The values of $SMALL$ and M are updated to correspond to the value and subscript of the smaller element. The remainder of

the elements in the array are then compared with SMALL. If a smaller element is found, SMALL and M are updated and the procedure continued.

Upon exit from this loop, the subscript of the smallest element is M and its value is SMALL. We then want to interchange element M with element J. The initial tendency is to use the statements

$$\begin{aligned} A_J &\leftarrow A_M \\ A_M &\leftarrow A_J \end{aligned}$$

Unfortunately, both elements J and M are equal to the original element M, which is *not* the desired result. Instead, the value of one of the elements must be stored in another storage location, say, TEMP, and used as follows:

$$\begin{aligned} \text{TEMP} &\leftarrow A_J \\ A_J &\leftarrow A_M \\ A_M &\leftarrow \text{TEMP} \end{aligned}$$

This accomplishes the desired result.

Although this procedure is used in the flowchart in Figure 4-4, the same result could be obtained with the following two statements:

$$\begin{aligned} A_M &\leftarrow A_J \\ A_J &\leftarrow \text{SMALL} \end{aligned}$$

The last value stored in variable SMALL was the element M of array A.

The flowchart in Figure 4-5 reads arrays STOCK and QUAN for the inventory problem, arranges STOCK in ascending order, and rearranges QUAN accordingly.

The Exchange Sort

Instead of scanning the array in search of the smallest element and then placing it first, suppose we scan the array by examining the elements two at a time, interchanging them whenever they are not in the desired order. The sequence of operations is illustrated in Table 4-2.

On the first pass, we begin by examining elements 1 and 2, which are 1407 and 2132, respectively. As they are in ascending order, no exchange is made. We then examine elements 2 and 3, which are 2132 and 1107. Since they are not in

Table 4-2. Progression of the Exchange Sort

Original array	After pass 1	After pass 2	After pass 3	After pass 4
1407	1407	1107	1107	1021
2132	1107	1407	1021	1107
1107	2132	1021	1407	1407
5072	1021	2132	2132	2132
1021	5027	5027	5027	5027

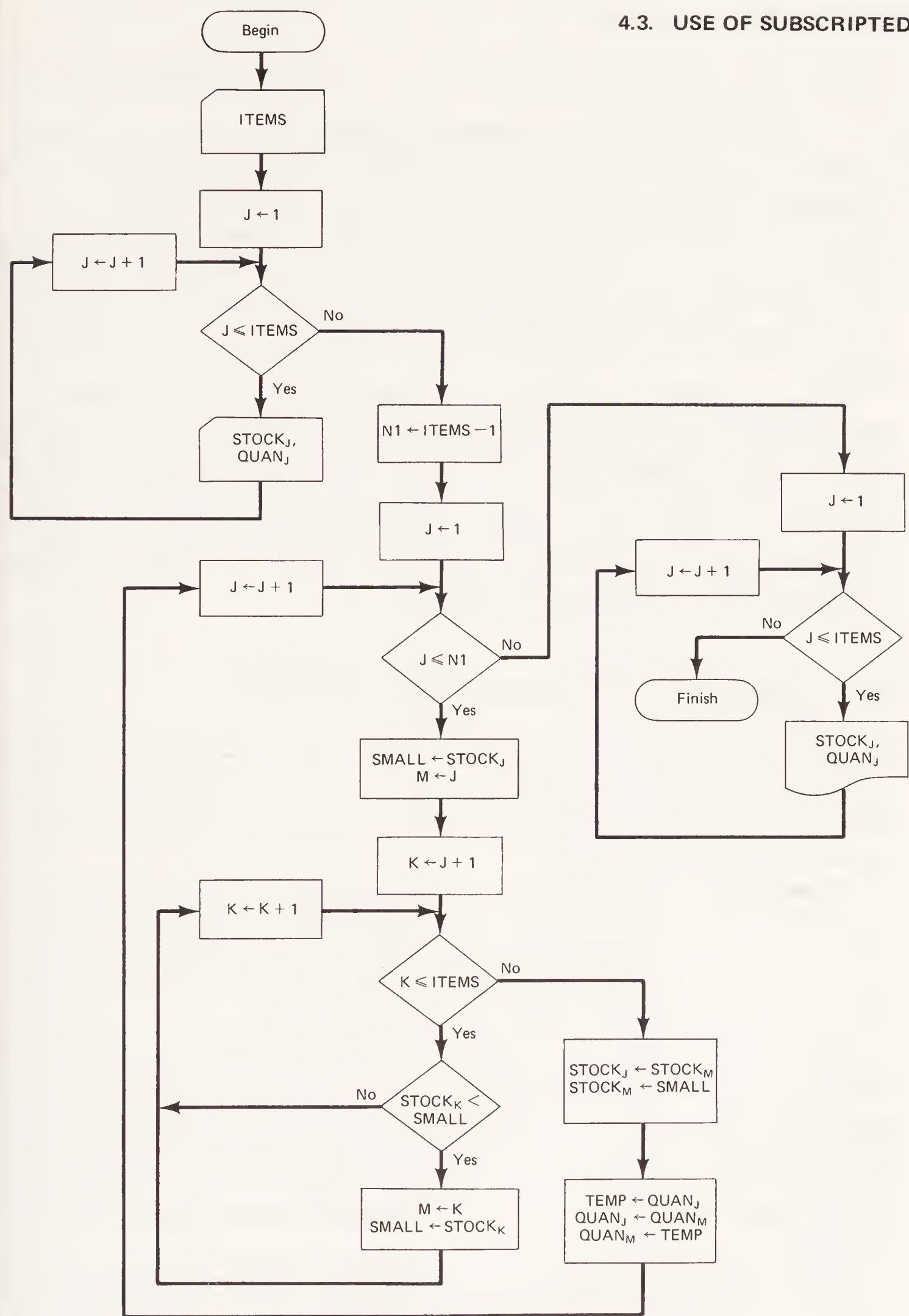


Figure 4-5 Arranging inventory in ascending order

ascending order, they are exchanged, giving 1407, 1107, 2132, 5072, and 1021 as the array. We then examine elements 3 and 4 (namely, 2132 and 5072), which are in ascending order and need not be exchanged. Examining elements 4 and 5

128 (namely, 5072 and 1021) indicates that they should be exchanged, giving the result shown in Table 4-2.

A cursory analysis of this problem should indicate a couple of the characteristics of this method. First, if the total number of entries in the array is n , the maximum number of passes required in the exchange sort will be $n - 1$. Furthermore, after the first pass, the largest entry must be in the last position. Therefore, there is no need to consider it any further. After the second pass, the two largest entries must be in ascending order in the last two positions and need not be considered in subsequent passes. This trend continues for the additional passes.

Although the above two observations could be used to devise an algorithm for the exchange sort, a better approach would be to record the index of the last element that was interchanged. If this were element k , then elements k through n are in the desired order. On the next pass, only elements up to and including element $k - 1$ need to be examined. Furthermore, if no elements or only the first two elements are exchanged on a particular pass, we are finished. This logic is incorporated in the flowchart of Figure 4-6.

4.4. TABLE LOOK-UP IN AN ORDERED ARRAY

In the flowchart in Figure 4-3, the only way to locate an element is to examine each one in turn. But by using an ordered array such as the output of the flowchart in Figure 4-5 or Figure 4-6, we may devise a more efficient algorithm for locating the desired element.

To understand the strategy to be used, suppose you want to obtain the telephone number for Joe Bleaux from the Manhattan telephone directory. Since it is in alphabetical order, you look a short way in from the front. For A. B. Kilroy, you look closer to the middle. That is, since it is an ordered list, you take advantage of the order to help you find the element.

But in doing this, you are taking advantage of your experience with the normal distribution of names beginning with certain letters. Although not very probable, a city could conceivably have very few people whose names begin with the letters L through Z. In this directory, A. B. Kilroy would be located close to the back instead of near the center. Can you devise a strategy that would work equally well in both the atypical situation and in the usual situation?

Suppose we begin as follows in searching for A. B. Kilroy. First, we open the directory near the center and examine the names. If we were extremely lucky, we would find A. B. Kilroy on this page. If not, all is not lost, since we can eliminate half of the telephone directory from the remainder of our search. For example, if the names at the center begin with letter L or higher, we can eliminate the last half of the directory. If they begin with J or lower, we can eliminate the first half.

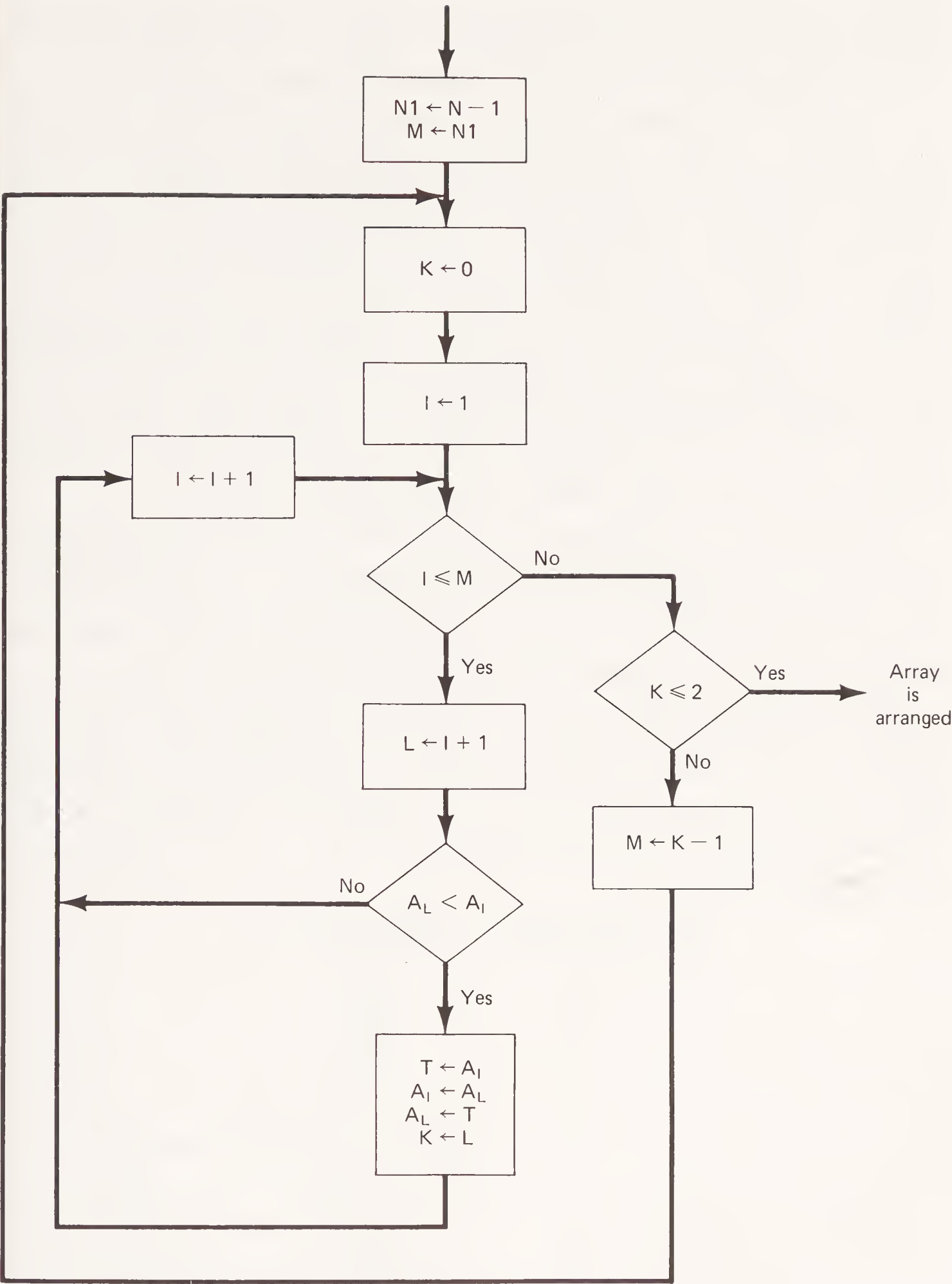


Figure 4-6 Flowchart of exchange-sort algorithm

We repeat this procedure with the remaining half. Again, we can eliminate another half of what is remaining, leaving us with a fourth of the original. This can be repeated as long as necessary and is surprisingly efficient. After we repeat the procedure ten times, the section known to contain A. B. Kilroy’s telephone

number is 1/1,024 of the original directory. That is, if the original directory contained 1,000 pages, we have narrowed our search down to a single page. We obtain the same result whether the distribution of names is normal or not.

A procedure very similar to this can be used to locate the stock numbers in the program of Figure 4-3. Suppose we have already read the current inventory. Locating the desired stock number can be accomplished via the flowchart in Figure 4-7. Initially, we know that the desired stock number must lie between elements 1 and ITEMS. Therefore, we assign the lower limit of our interval, LOWER, the value of 1 and the upper limit, UPPER, the value of ITEMS. The subscript of the center element on this interval is computed and the value of the element is compared to STKNO. If equal, the desired entry has been located. If STOCK_M is less than the STKNO, the desired entry must be between M and UPPER. We therefore update LOWER by assigning it the value of M. If STOCK_M is greater than STKNO, the desired entry must lie between LOWER and M. We therefore update UPPER by assigning it the value of M.

This procedure can be repeated as many times as desired. If there is an entry in array STOCK that equals STKNO, it will be located. If not, eventually UPPER and LOWER will differ by only one. In this case, the same value of M will be used on each iteration. We can therefore stop whenever the computed value of M equals the value used on the previous iteration.

Let's compare the efficiency of this approach with that of the sequential search used in Figure 4-3. Suppose our inventory contains 1,000 items, and suppose we enter a stock number that does not exist. How many comparisons shall we make in each case? For the sequential search, we shall make 1,000 comparisons. For the interval halving approach, the interval of uncertainty is decreased as follows:

Iteration	Interval
0	1000
1	500
2	250
3	125
4	63
5	32
6	16
7	8
8	4
9	2
10	1

Therefore, only ten comparisons are needed to conclude that the desired entry does not exist in array STOCK. In fact, you can show that ten comparisons are sufficient to locate any element in array STOCK via this method,

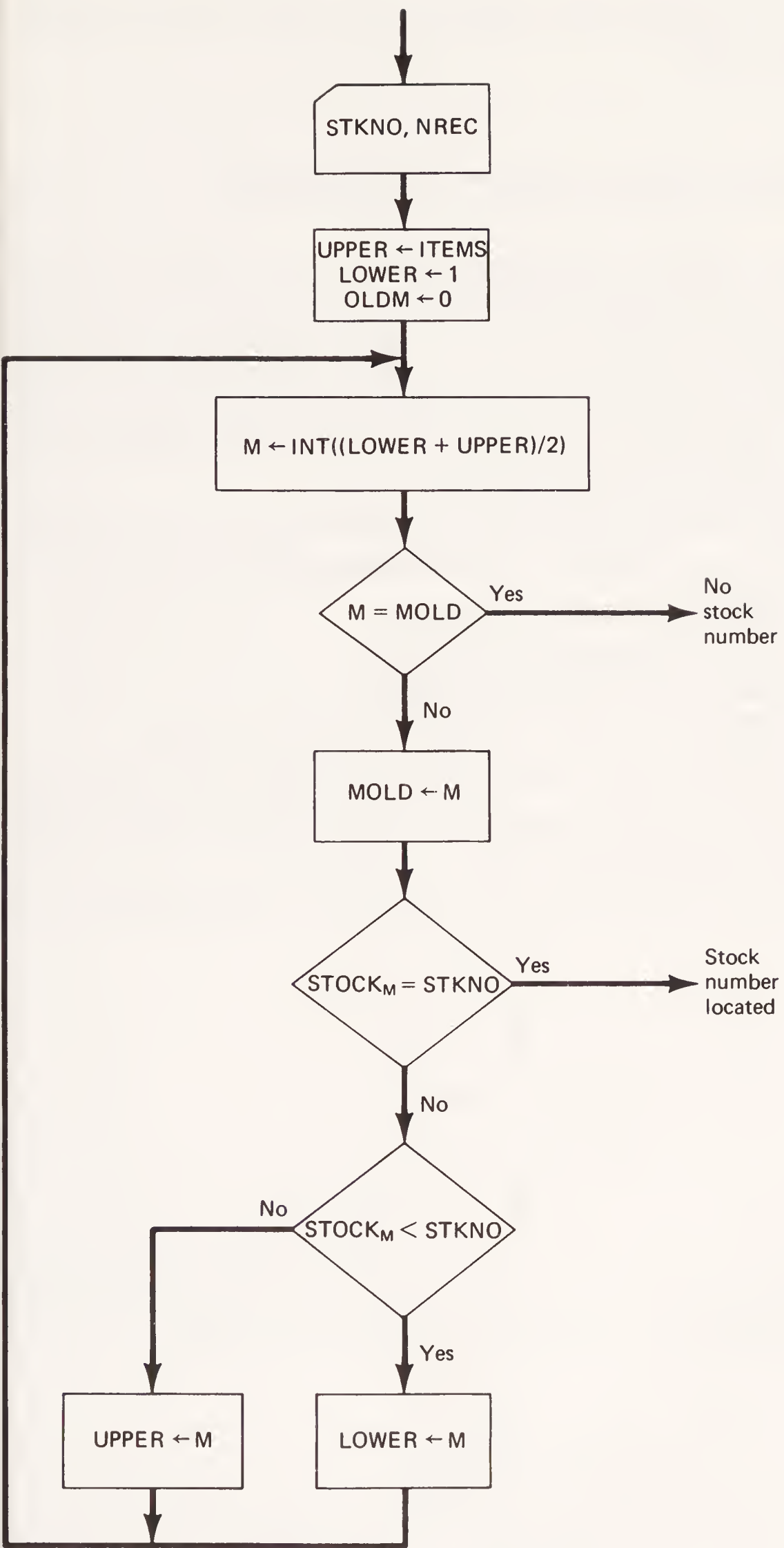


Figure 4-7 Locating entries in an ordered array

132 as compared with an average of 500 comparisons using the sequential method. This is a substantial savings.

4.5. INPUT/OUTPUT STATEMENTS FOR ARRAYS

In the flowcharts presented in previous sections of this chapter, a READ statement was placed inside a loop so that the elements of an array could be read one by one. On output, PRINT statements were placed inside a loop in a similar fashion. An alternative to this is to place a loop within the input/output statement, two examples being given in Figure 4-8. The READ example calls for the values of A_1 , A_3 , A_5 , A_7 , A_9 , and A_{11} to be entered. The output example calls for the values of B_1 , B_2 , B_3 , and B_4 to be printed.

The general form of the statement inside the input/output block is as follows:

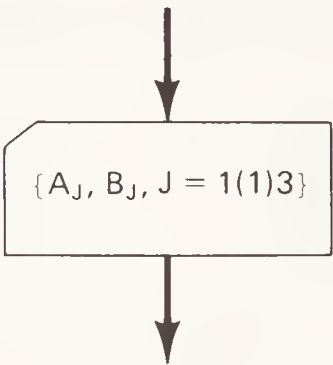
where

$$\{A_{INDEX}, INDEX = INIT(INCRA)LIMIT\}$$

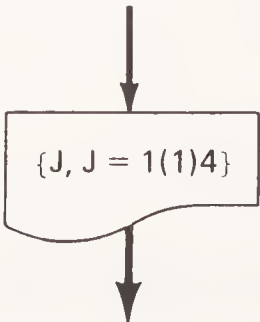
INDEX is the index of the loop
INIT is the initial value of INDEX
LIMIT is the final value of INDEX
INCRA is the value by which INDEX is incremented

Expressions may be used in place of the indexing parameters INIT, INCRA, and LIMIT. We will place no more restrictions on these than were placed on the indexing parameters of the iteration counter.

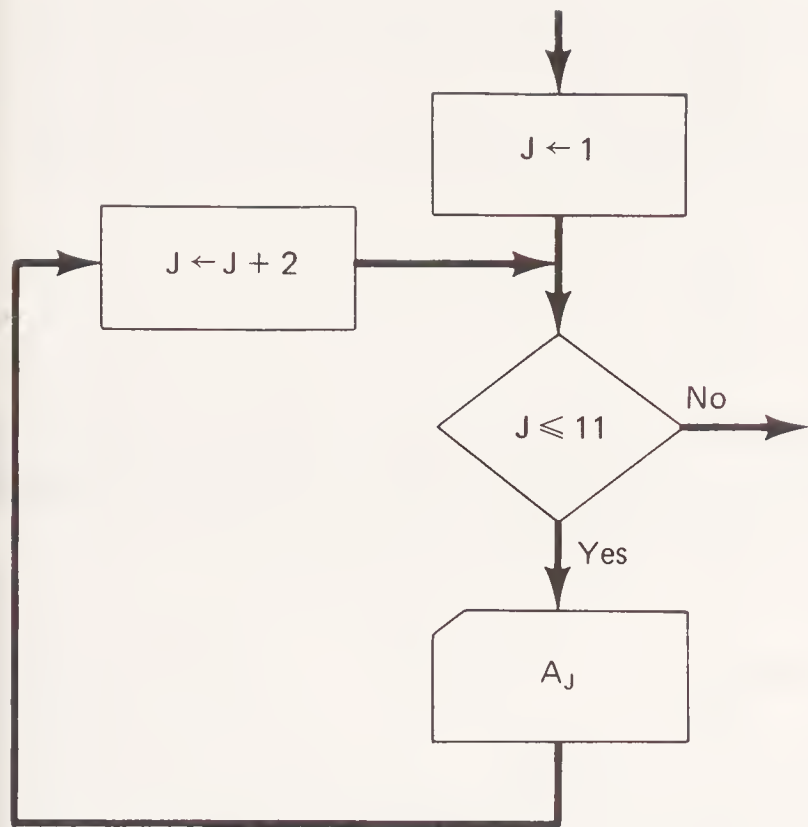
More than one array may be used in this type of input/output statement. For example, the block



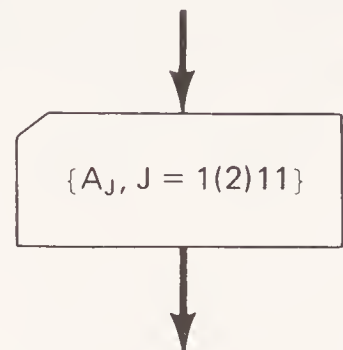
reads values in the sequence $A_1, B_1, A_2, B_2, A_3, B_3$. It is also permissible to use the index in ways other than as a subscript. For example, the block



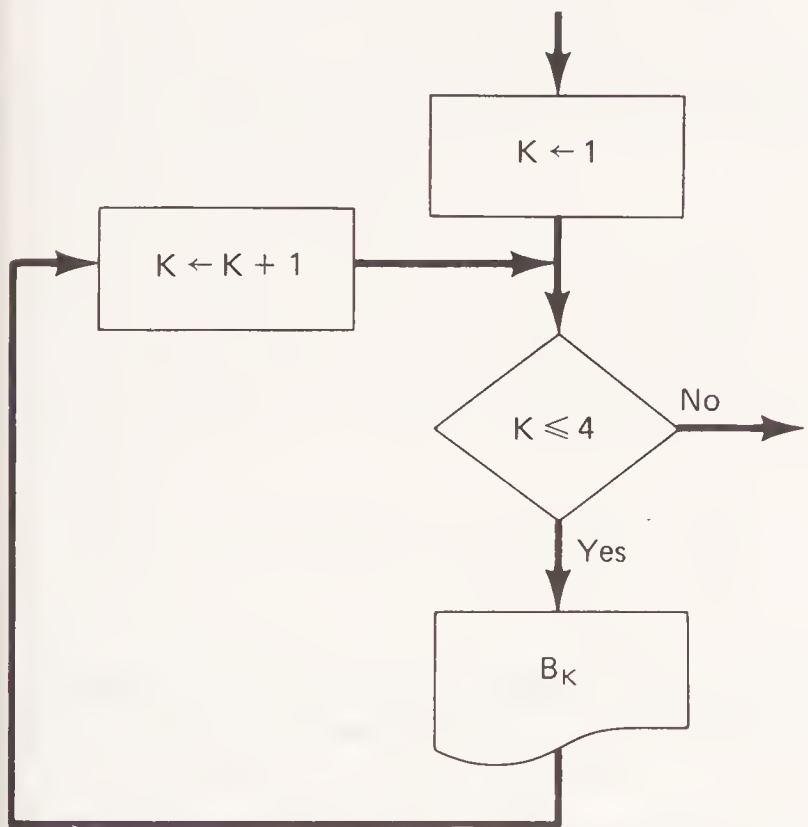
prints the numbers 1, 2, 3, 4. A commonly used block is



a. READ block inside a loop

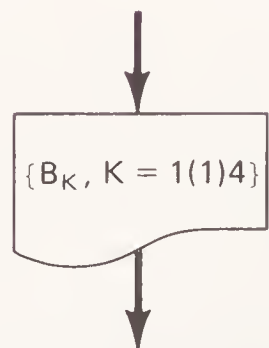


b. Loop inside a READ block

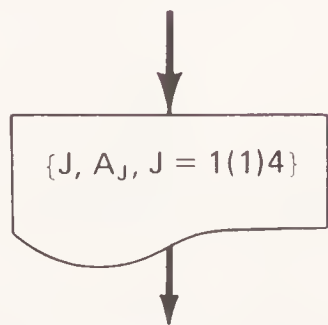


c. Output block inside a loop

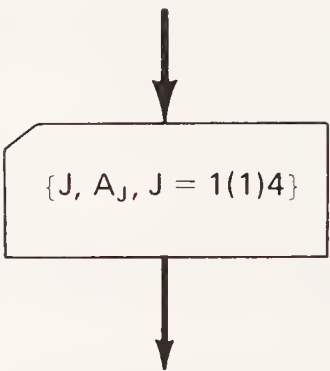
Figure 4-8 Input/output of arrays



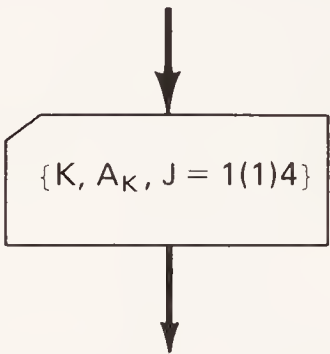
d. Loop inside an output block



which prints the subscript number along with the corresponding value. The corresponding input block

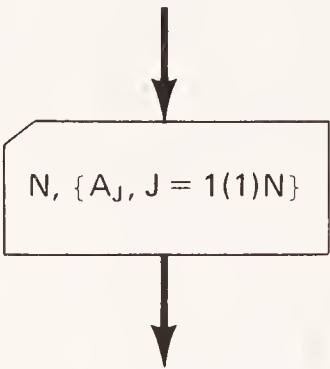


is inappropriate, since reading a value for J would not be consistent with the incrementing procedure specified by the indexing parameters. To read a value of a subscript and the corresponding value, the appropriate input block is



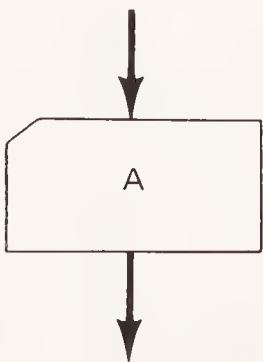
The index is not altered by the input values.

It is also permissible to enter the total number of entries followed by their values via the block



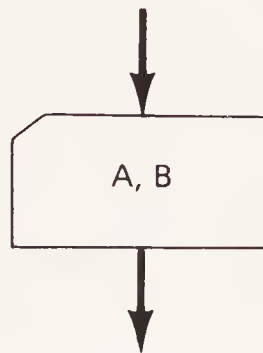
Values are assigned to variables in the order in which they appear, which permits these values to be used as either subscripts or indexing parameters in subsequent portions of the input block.

In the previous examples, a subscript has always accompanied the name of the array. In input/output statements, the name of the array may be used without being accompanied by a subscript, for example,



This literally means “read array A.” If array A has been specified to contain 20 elements, then 20 values are read. The first value entered is stored in A_1 , the second in A_2 , etc.

More than one array name may appear in a block, for example,



The entire array A is read followed by array B. Output blocks are treated in an analogous manner.

In many programming languages, the arrangement in Figure 4-8b is only a more convenient equivalent to the arrangement in Figure 4-8a. In others, notably Fortran, there is a difference, arising from the fact that input and output data is arranged in the form of records. For example, for card input the 80 characters on the card constitute a record from which numerical values can be extracted. Similarly, on output a record consists of the characters that compose a single line. Numerical values and other information can be used to construct these characters.

Each time an input or output block is executed, a *new* record is processed. Therefore, the arrangement in Figure 4-8a requires that the value of A_1 be taken from the first record, the value of A_3 from the second, the value of A_5 from the third, etc. In all, a total of six records are processed, a single value being read from each.

Using the arrangement in Figure 4-8b, the READ block is executed only once. Therefore, the values for $A_1, A_3, \dots, A_{11}$ could all be read from the same record. In this case, the loop is inside the READ block, whereas in Figure 4-8a the READ block is inside the loop. While additional features often permit use of the arrangement in Figure 4-8b to extract a single value from each of six records, the arrangement in Figure 4-8a cannot be used to read all values from the same record. In short, the arrangement in Figure 4-8b can usually be used to accomplish anything possible using the arrangement in Figure 4-8a, but the converse is not true.

Analogous statements also apply to the output blocks in Figure 4-8c and 4-8d.

4.6. USE AND MISUSE OF SUBSCRIPTED VARIABLES

While a large number of problems can be conveniently solved only with the use of subscripted variables, other problems can be solved either with or without

subscripted variables. The main disadvantage of using subscripted variables is that the unnecessary storage of large arrays needlessly consumes memory, which is frequently the resource in shortest supply. In other cases; a problem can be more efficiently programmed using subscripted variables than could be done otherwise. In this case, the question of importance is, “Is the reduction in machine time required justified by the additional memory required?” Since many centers charge according to the running time of the program and the amount of memory required, the decision is ultimately economic in nature, although a formal analysis is made only for very large problems.

In this section, we shall present a couple of examples of algorithms that utilize subscripted variables to solve problems in which they really are not needed at all. Students should certainly avoid the use of subscripted variables when no benefit accrues.

Example 1. Fibonacci Sequence

In Section 3.3, we presented an algorithm for computing the Fibonacci sequence without using subscripted variables. An alternative approach is to use the algorithm in Figure 4-9. Although this algorithm is perhaps a little easier to visualize than the one in Figure 3-9, it is by no means significantly superior in any other respect.

Example 2. Printing a Tabular Function

Suppose in our day-to-day work we frequently need values of the function

$$f(x) = \frac{x + 1}{x^2 + 4(x + 1) + \sqrt{x}}$$

The variable x will always be greater than or equal to zero. Values of x exceeding 140 are rarely encountered. To eliminate much of the effort in evaluating this function, the boss has requested his computing staff to devise a set of tabular values for this function. For ease of use, he wants the output to appear as shown in Figure 4-10.

Several approaches could be used to solve this problem, some of which use subscripted variables to varying degrees. One approach would be via the algorithm in Figure 4-11a. In this algorithm, the entire set of numbers in the table are computed and stored in arrays X and F , which are then printed in such a fashion that three columns result in the output. With this approach, at least 150 storage locations must be reserved for each of arrays X and F .

For this problem, there is no reason that the entire arrays must be stored. At most, only the contents of a single line of the output must be stored. That is, we shall compute all entries on line 1 of the output, print them, and then proceed to the next line of output. In this way, we can significantly reduce storage requirements.

One approach is to use arrays, again X and F , to store the values for each line

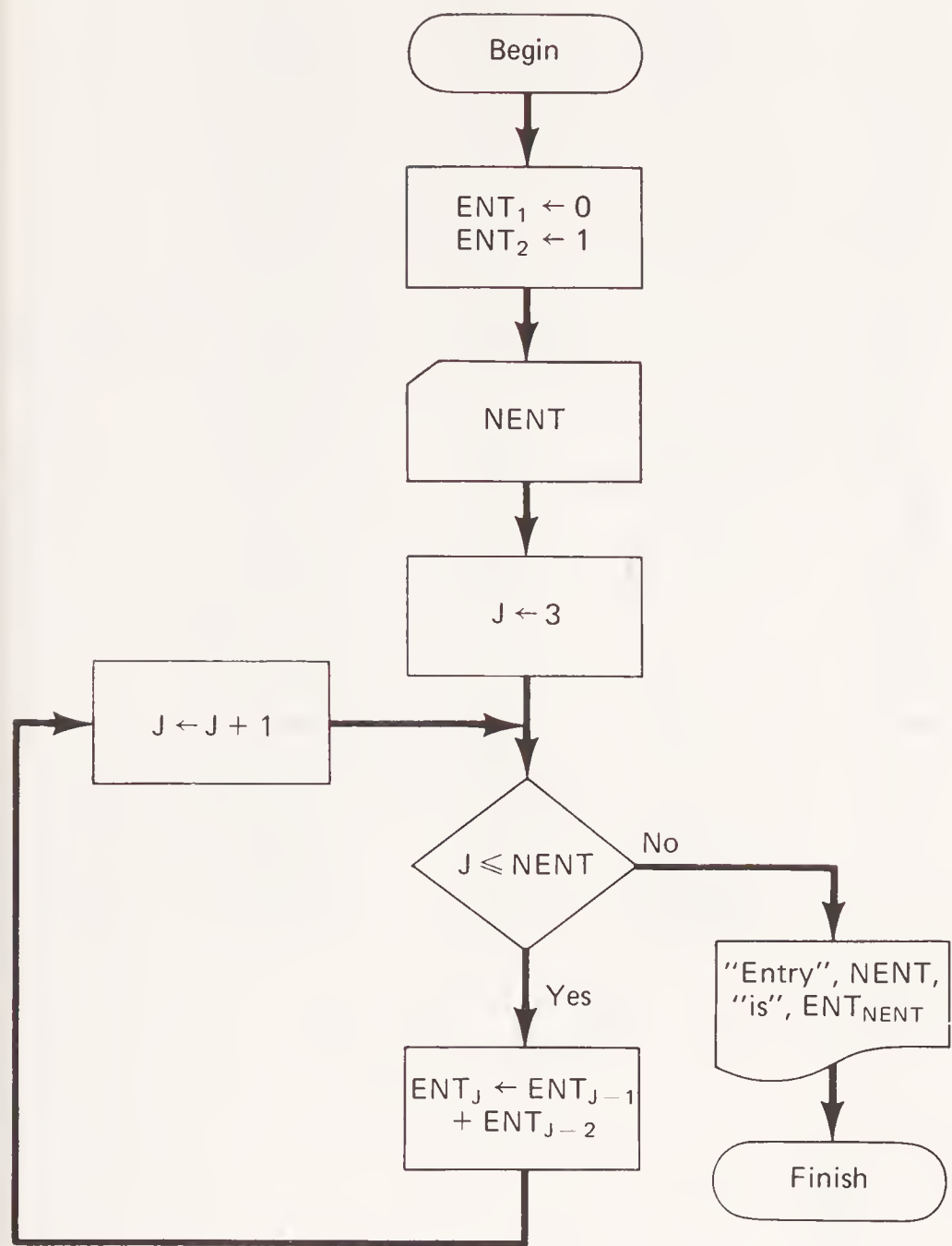


Figure 4-9 Flowchart of algorithm to compute Fibonacci sequence

of output. The numbers are computed, stored, and then printed as illustrated by the flowchart in Figure 4-11b. In this case, only three storage locations are reserved for arrays X and F.

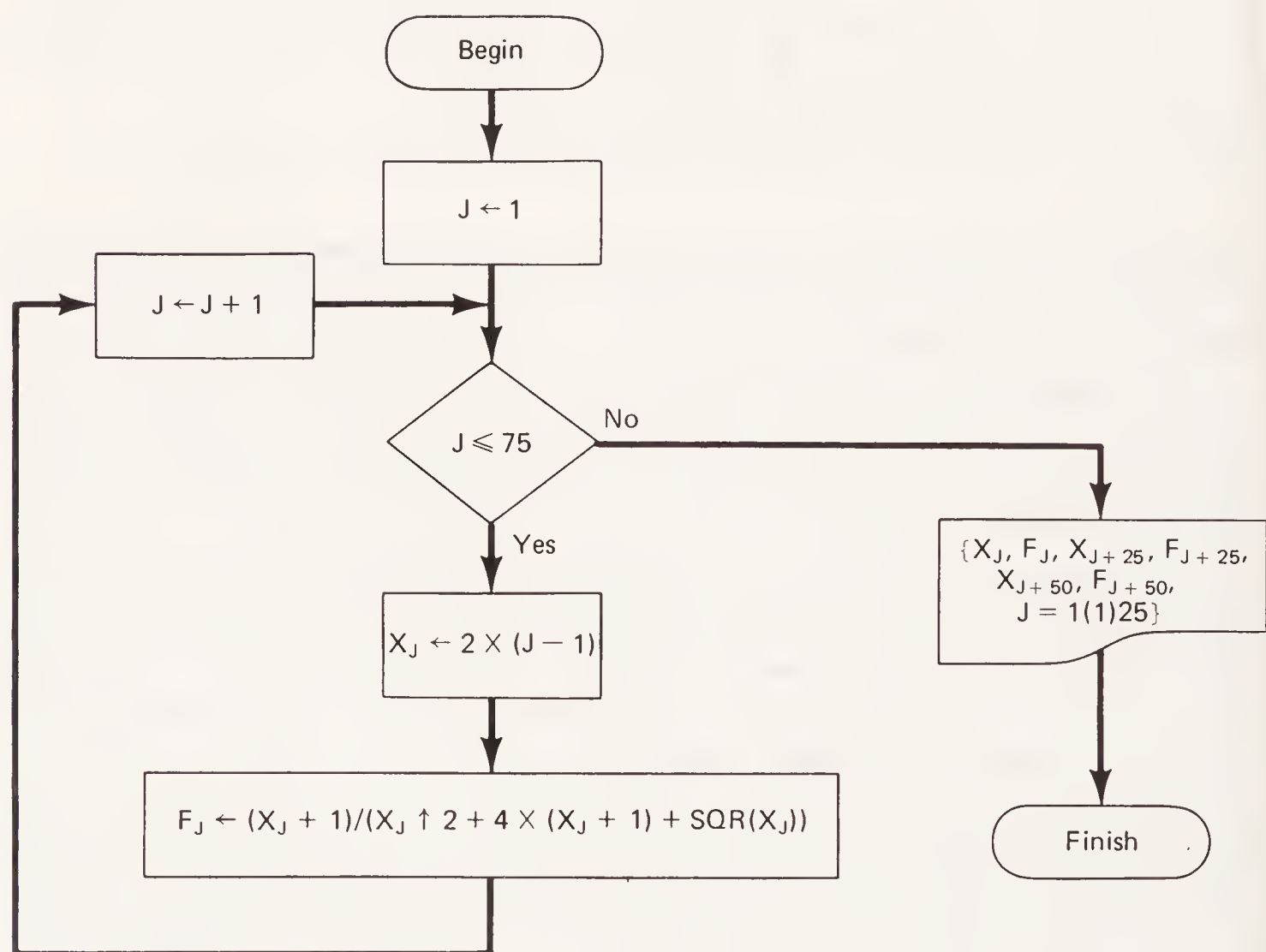
Since these arrays are quite small, they can be eliminated by essentially using separate statements to accomplish the computations made on each pass around the inner loop of Figure 4-11b. In fact, the flowchart in Figure 4-11c is much simpler than the one in Figure 4-11b.

In programming languages such as Basic that permit part of an output line to be printed at one time and then added to in successive printings, the arrays can be eliminated from the flowchart in Figure 4-11b, provided that the output block is placed within the inner loop. But since this capability is not within the scope of our output block, this option is not open to us.

Many practical problems are similar to this problem in that the program basically depends upon the complexity of the problem but not on the number of data cases. Computing the sum of a set of numbers is a simple but good example. A loop is constructed that adds each entry to a running sum. A single entry in the data set is processed upon each execution of the statements within the loop.

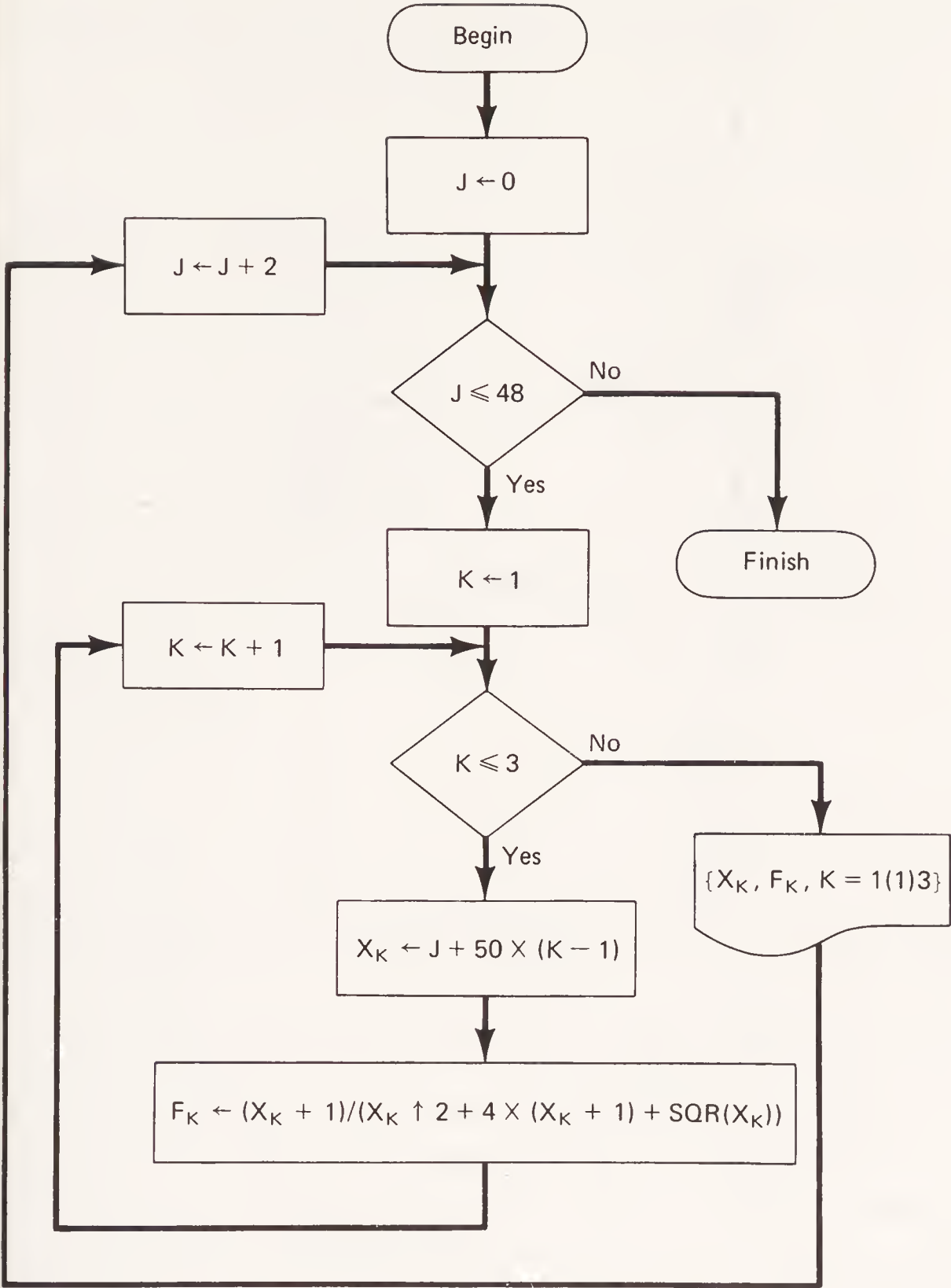
X	F(X)	X	F(X)	X	F(X)
0.0	0.25000	50.0	0.001881	100.0	0.00970
2.0	0.17227	52.0	0.001813	102.0	0.00951
4.0	0.13158	54.0	0.001750	104.0	0.00934
6.0	0.10534	56.0	0.001691	106.0	0.00917
8.0	0.08752	58.0	0.001635	108.0	0.00900
10.0	0.07475	60.0	0.001584	110.0	0.00884
12.0	0.06517	62.0	0.001535	112.0	0.00869
14.0	0.05775	64.0	0.001489	114.0	0.00854
16.0	0.05183	66.0	0.001446	116.0	0.00840
18.0	0.04700	68.0	0.001406	118.0	0.00826
20.0	0.04299	70.0	0.001367	120.0	0.00812
22.0	0.03961	72.0	0.001331	122.0	0.00799
24.0	0.03672	74.0	0.001297	124.0	0.00787
26.0	0.03422	76.0	0.001264	126.0	0.00775
28.0	0.03203	78.0	0.001233	128.0	0.00763
30.0	0.03011	80.0	0.001203	130.0	0.00751
32.0	0.02841	82.0	0.001175	132.0	0.00740
34.0	0.02689	84.0	0.001148	134.0	0.00729
36.0	0.02552	86.0	0.001122	136.0	0.00719
38.0	0.02428	88.0	0.001097	138.0	0.00709
40.0	0.02316	90.0	0.001074	140.0	0.00699
42.0	0.02214	92.0	0.001051	142.0	0.00689
44.0	0.02120	94.0	0.001030	144.0	0.00680
46.0	0.02034	96.0	0.001009	146.0	0.00671
48.0	0.01955	98.0	0.00989	148.0	0.00662

Figure 4-10 Tabulation of function $f(x) = \frac{x + 1}{x^2 + 4(x + 1) + \sqrt{x}}$



a. Use of large arrays

Figure 4-11 Algorithms for computing a tabular function



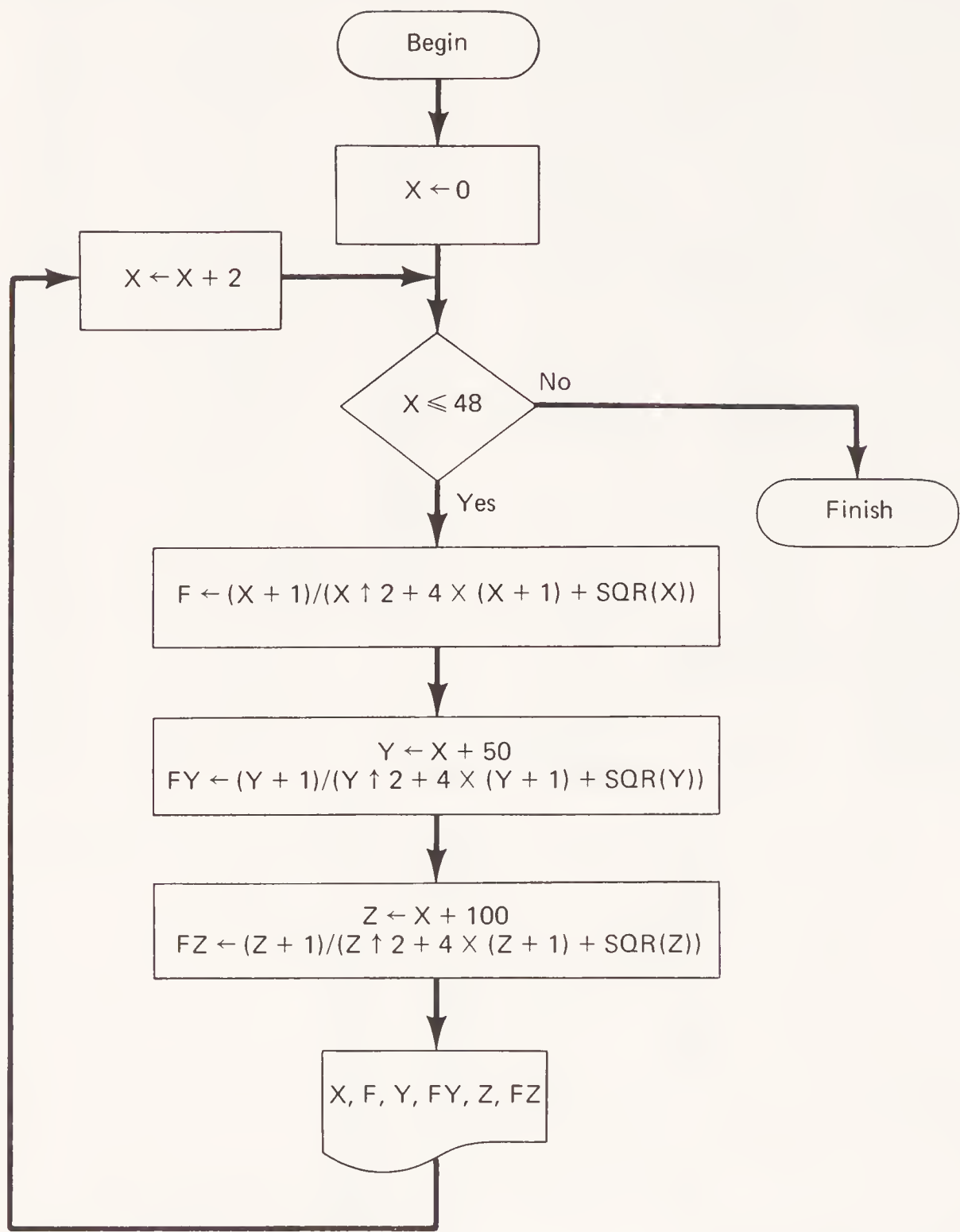
b. Use of small arrays

Figure 4-11 (Continued)

The complexity of the statements within the loop depends upon what is to be done with each entry, not upon how many entries are in the set. When these entries are not needed beyond the loop in which they are processed, there is no need to store each value in a separate storage location, and therefore no need for arrays.

4.7. ARRAYS WITH MULTIPLE SUBSCRIPTS

The concept of the single-subscripted array presented in previous sections of this chapter can be readily extended to arrays with more than one subscript. Although most programming languages permit arrays to have more than two



c. No use of arrays

Figure 4-11 (Continued)

subscripts, in this section we shall restrict our attention to double-subscripted arrays. Once the reader understands these, he can readily extend the concept to arrays with more subscripts. We shall begin with an example problem and discuss how it is solved using arrays with double subscripts.

Suppose at the end of the term, an instructor has accumulated five exam scores on each of his students, as illustrated in Table 4-3a. He would like to compute the following quantities:

- 1. Average score on each exam.
- 2. Average score made by each student over the five exams.
- 3. Average score made on all quizzes.

Upon completion of these computations, the summary illustrated in Table 4-3b is to be produced for each student.

Table 4-3. Data and Output for Exam-Scores Example

A. EXAM SCORES

Student	Name	EXAM				
		1	2	3	4	5
1	Joe Bleaux	76	80	66	93	90
2	John Doe	82	88	98	100	95
3	A. B. Kilroy	72	70	88	72	100
4	Joe Kollage	91	70	88	80	86
5	Z. Phantom	70	75	61	67	80
6	Abe Ringer	65	65	75	82	75
7	John Smith	90	96	68	65	83

B. DESIRED OUTPUT—STUDENT 1

Exam	Score	Class average
1	76	78.0
2	80	77.8
3	66	77.8
4	93	79.9
5	90	87.0

Student’s average on all exams 81.0

Class average on all exams 80.1

In order to efficiently compute these values, the first step is to enter the individual scores into memory. One approach could be to define five single-subscripted arrays, each of which would contain the scores made by all students on one of the exams. This approach basically suffers the same disadvantages enumerated in Section 4.1 for using simple variables to store the ages to be averaged. For our problem, use of a double-subscripted array is more convenient than use of single-subscripted arrays.

In this example, we will use a double-subscripted array SCORE to store the scores in Table 4-3. These scores are arranged in a table, which is said to be composed of rows and columns. In Table 4-3a each column contains the score made by all students on one of the exams. Each row contains the score made by a single student on all exams. An element in the table can be located by designating its row number and column number. That is, on exam 2 (that is, column 2) Joe Kollage (that is, row 4) made a score of 70.

In working with double-subscripted arrays, we shall use the same approach for storing and retrieving entries. According to mathematical convention, the first subscript designates the row and the second designates the column. By this convention, the element on row J and column K of our table will be designated SCORE_{J,K}. Although we shall adopt this designation throughout the text, the computer does not depend upon this, and the opposite designation could be used as long as it is consistently followed throughout the program.

To enter elements into this array, we could read the scores by columns, as illustrated in Figure 4-12a. For input by cards, the data would be arranged as follows:

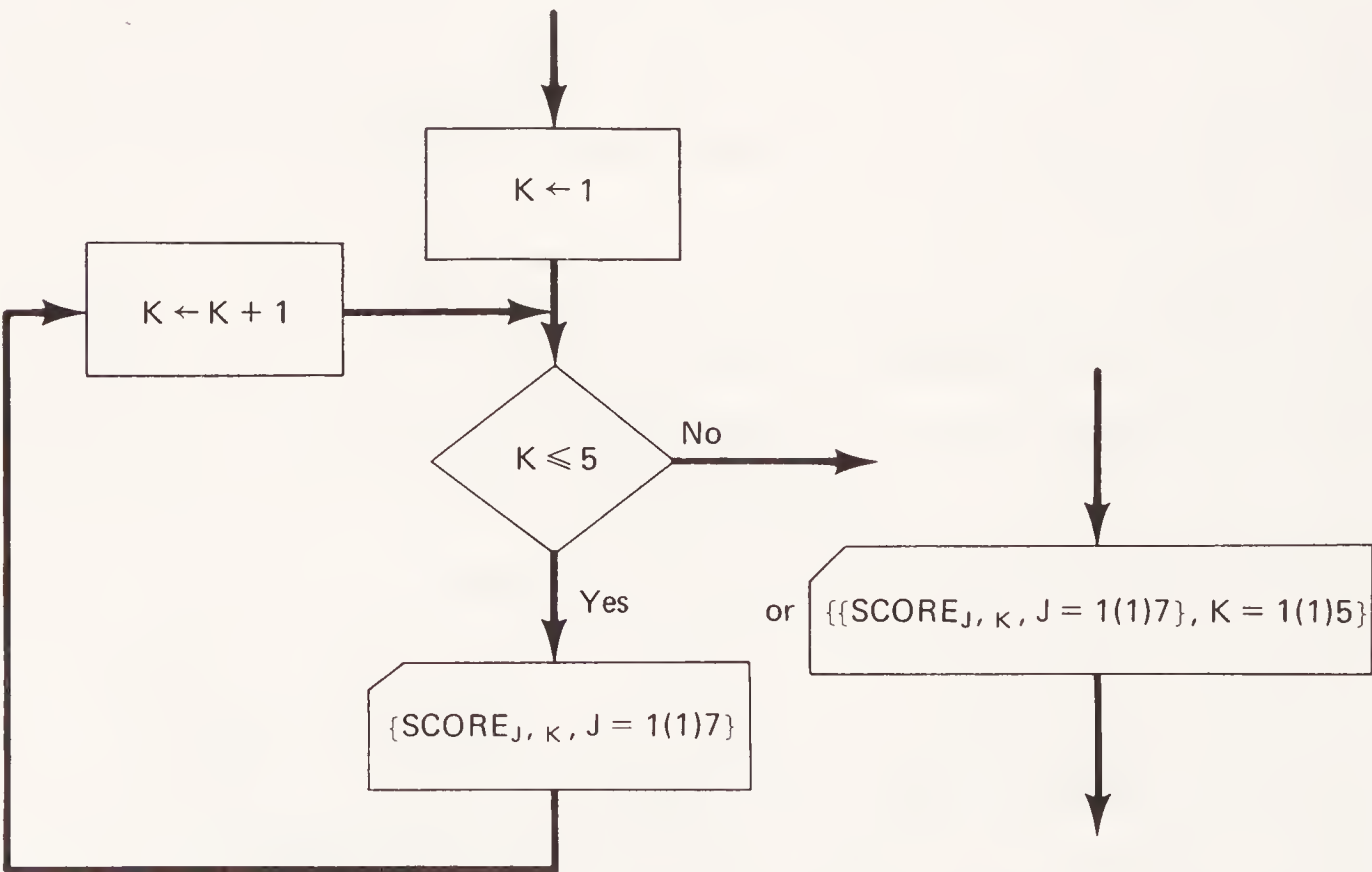
- Card 1: 76, 82, 72, 91, 70, 65, 90
- Card 2: 80, 88, 70, 70, 75, 65, 96
- Card 3: 66, 98, 88, 88, 61, 75, 68
- Card 4: 93, 100, 72, 80, 67, 82, 65
- Card 5: 90, 95, 100, 86, 88, 75, 83

Alternatively, the elements could be entered by rows, utilizing the arrangement of Figure 4-12b. For card input, the data arrangement would be

- Card 1: 76, 80, 66, 93, 90
- Card 2: 82, 88, 98, 100, 95
- Card 3: 72, 70, 88, 72, 100
- Card 4: 91, 70, 88, 80, 86
- Card 5: 70, 75, 61, 67, 88
- Card 6: 65, 65, 75, 82, 75
- Card 7: 90, 96, 68, 65, 83

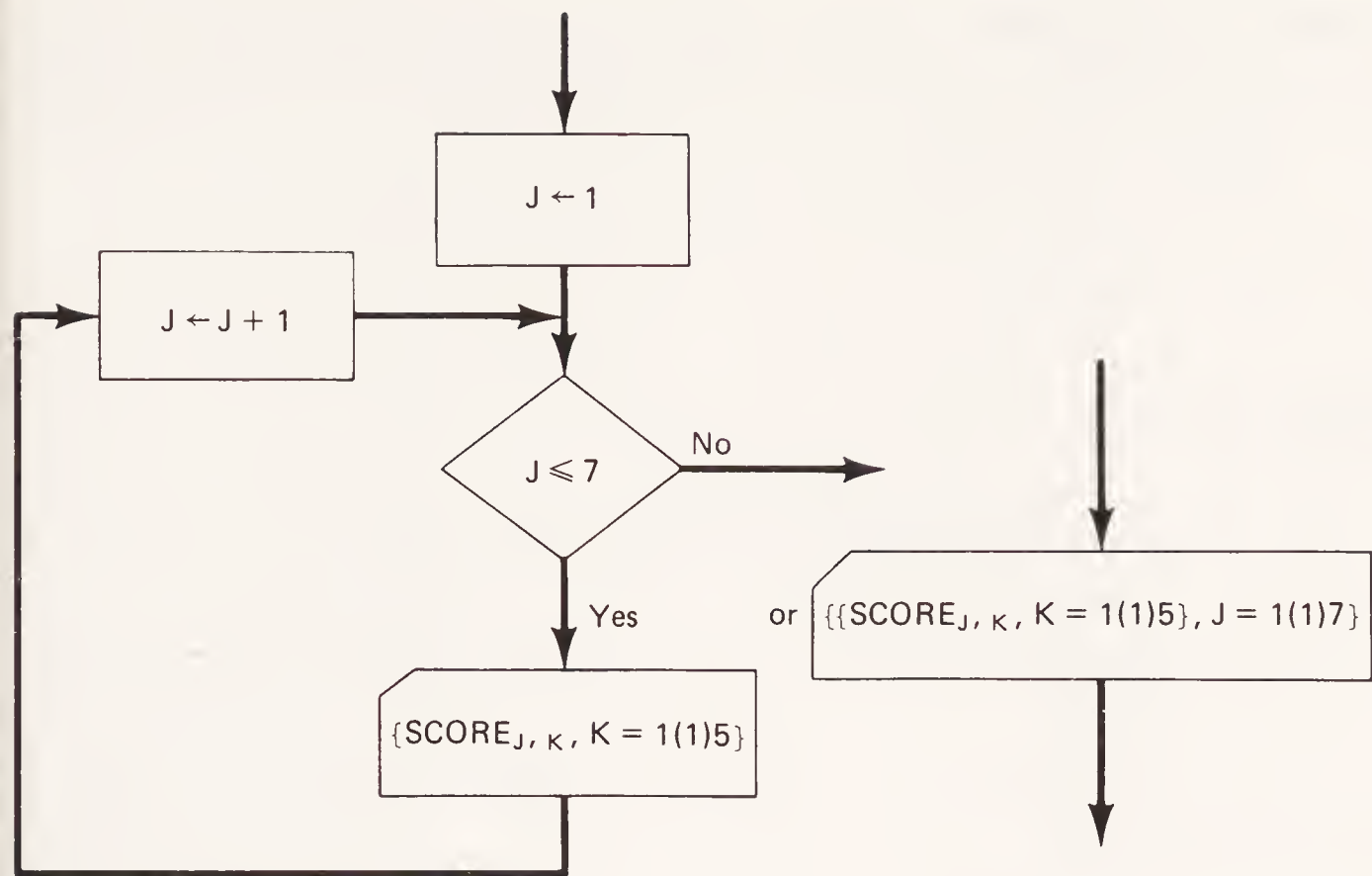
Either of these alternatives is acceptable, provided the data arrangement corresponds to the arrangement of input statements. The special statement for input of arrays is extended to double-subscripted arrays in Figure 4-12.

The flowchart for this problem is given in Figure 4-13. The scores are first entered by rows, which is followed by the computation of the average score on each exam (stored in the single-subscripted array EXAVG) and the average of all



a. Reading the scores by columns

Figure 4-12 Reading the scores in Table 4-3



b. Reading the scores by rows
Figure 4-12 (Continued)

scores (AVG). The final section of the flowchart computes the average score on all exams for each student and prints the desired output. Note that there is no need to use an array to store the average score of each student, since the result is printed immediately after being computed.

In this particular example, the specifications on the printed output is the only requirement that necessitates the use of a double-subscripted array. If only the average score on each exam, the average score for each student, and the overall average are to be printed, the use of a double-subscripted array can be avoided by using the flowchart in Figure 4-14. Instead of storing the entire table in Table 4-3a via a double-subscripted array, a row (that is, scores of all students on a single exam) is entered and stored in the single-subscripted array GRADE. Another single-subscripted array EXAVG is first used to accumulate totals of all scores on each exam. This total score is converted to an average score just prior to printing. The scores made by the student are summed, divided by 5 to obtain the average, and stored in the single-subscripted array STAVG for subsequent printing. This flowchart avoids storage of the complete table and is probably no more difficult to program than the one given in Figure 4-13 that uses a double-subscripted array.

4.8. PROCESSING OF CHARACTER DATA

Up to this point in the text, we have been almost exclusively concerned with numerical problems. To a certain extent, many aspects of the processing of nonnumerical data could have been presented in earlier chapters. We have chosen

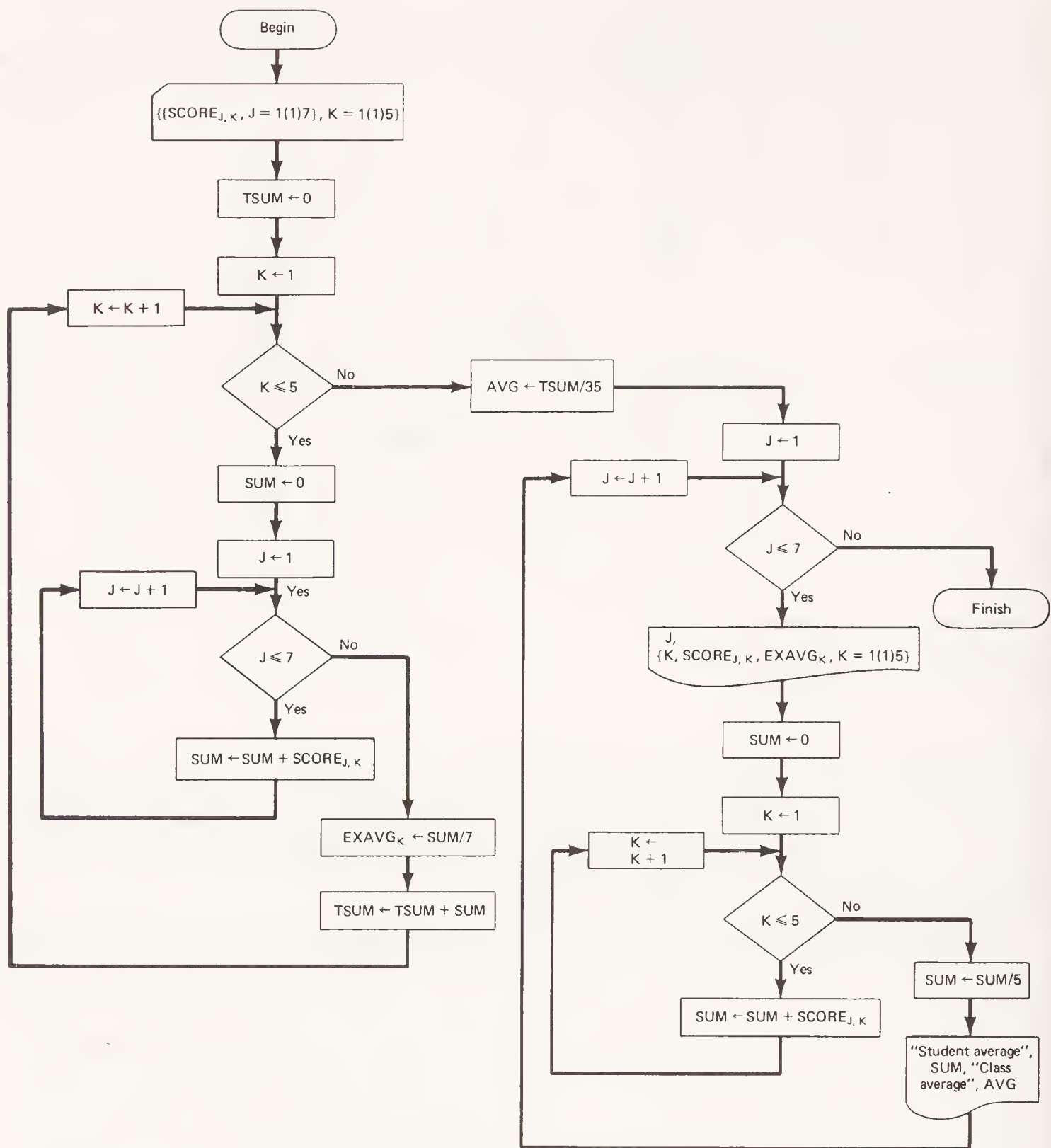


Figure 4-13 Flowchart for processing the scores in Table 4-3

not to introduce it until this point in order that the discussion would appear at a single point rather than be distributed throughout the text.

In our use of the term, *character data* may consist of any character recognizable by the computer system being used. These include the ten digits, the twenty-six letters of the alphabet (some systems distinguish between upper and lower case), the blank or space character, and several additional characters (decimal point, comma, parentheses, etc.). The total number of characters available generally ranges from as low as 48 to a maximum of 256. The term *alphanumeric* or *alphanumeric* is frequently used in lieu of our term *character data*. Furthermore, the term *alphabetic data* is often used to apply to a subset of character data consisting of the letters of the alphabet.

Character data can be stored in the computer's memory in much the same

fashion that numerical values are stored. Similarly, variables can be used with character data. For example, suppose we want to store the following sentence in memory:

DAD IS SAD.

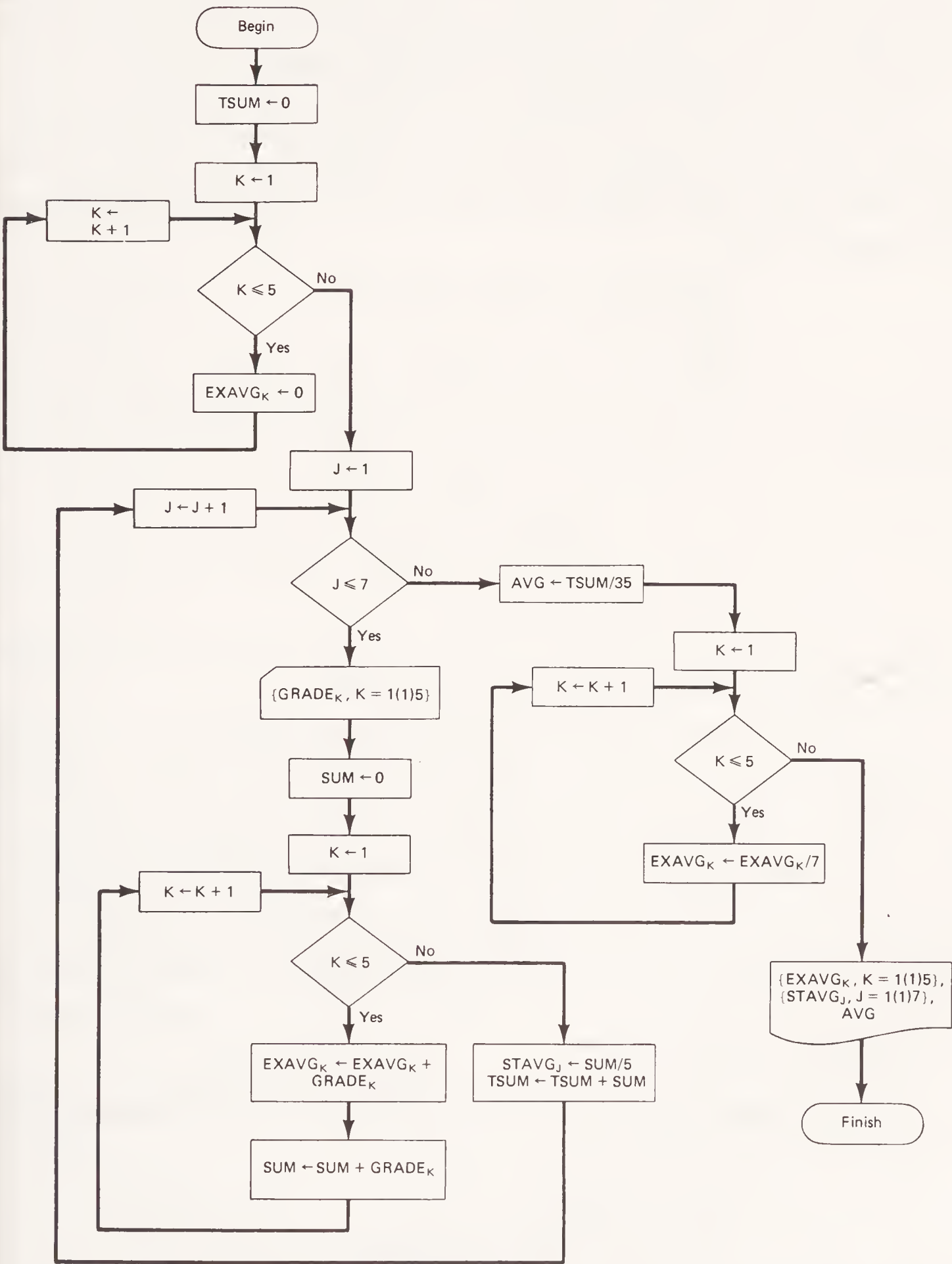


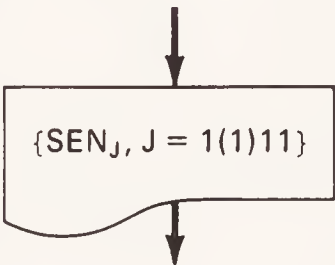
Figure 4-14 Flowchart for processing the scores in Table 4-3 without using a double-subscripted array

Such sequences of associated characters are often referred to as *strings*. Counting the two blank spaces and the period, this string consists of eleven characters. We might choose to store these characters in the area in memory reserved for a simple variable, say, SENTENCE. Alternatively, we might choose to use a single-subscripted array, say, SEN, and store the sentence character by character. That is, store the character D in the location reserved for SEN₁, store the character A in the location reserved for SEN₂, etc. Which approach we choose will depend upon what type of processing we want to perform and the character-manipulating facilities available in the programming language employed.

Although the arithmetic operations of addition (+), subtraction (−), multiplication (×), division (/), and exponentiation (↑) are meaningless when applied to character data, the replacement operation (←) applies to character data in much the same fashion that it applies to numeric data. For example, the following statements are valid:

```
SENTENCE ← "DAD IS SAD."  
SEN1 ← "D"  
SEN2 ← "A"
```

Input blocks can also be used to assign character data to variables. For example, the block



would read the sentence given above and store one character in the storage location reserved for each element of the array SEN.

In addition to the replacement operation, the relational operators or their equivalent are valid for character data. While many systems employ a collating sequence as a basis for performing “greater than” and “less than” comparisons, we shall be interested only in the relational operators “equal to” and “not equal to.” For example, suppose you want to determine the number of appearances of the character “A” in the sentence we have been using in our discussion. In the flowchart in Figure 4-15, each character in the sentence is compared with “A”, and a counter incremented upon equality. This approach could be extended to determine such information as the number of words in a sentence and the average length of the words used.

As some operations are defined for numeric data but not character data, there are operations defined for character data but not for numeric data. For example, suppose we have the following two strings:

```
STR1 ← "I AM"  
STR2 ← "GOING TO TOWN"
```

The concatenation operation consists of linking these two strings to form

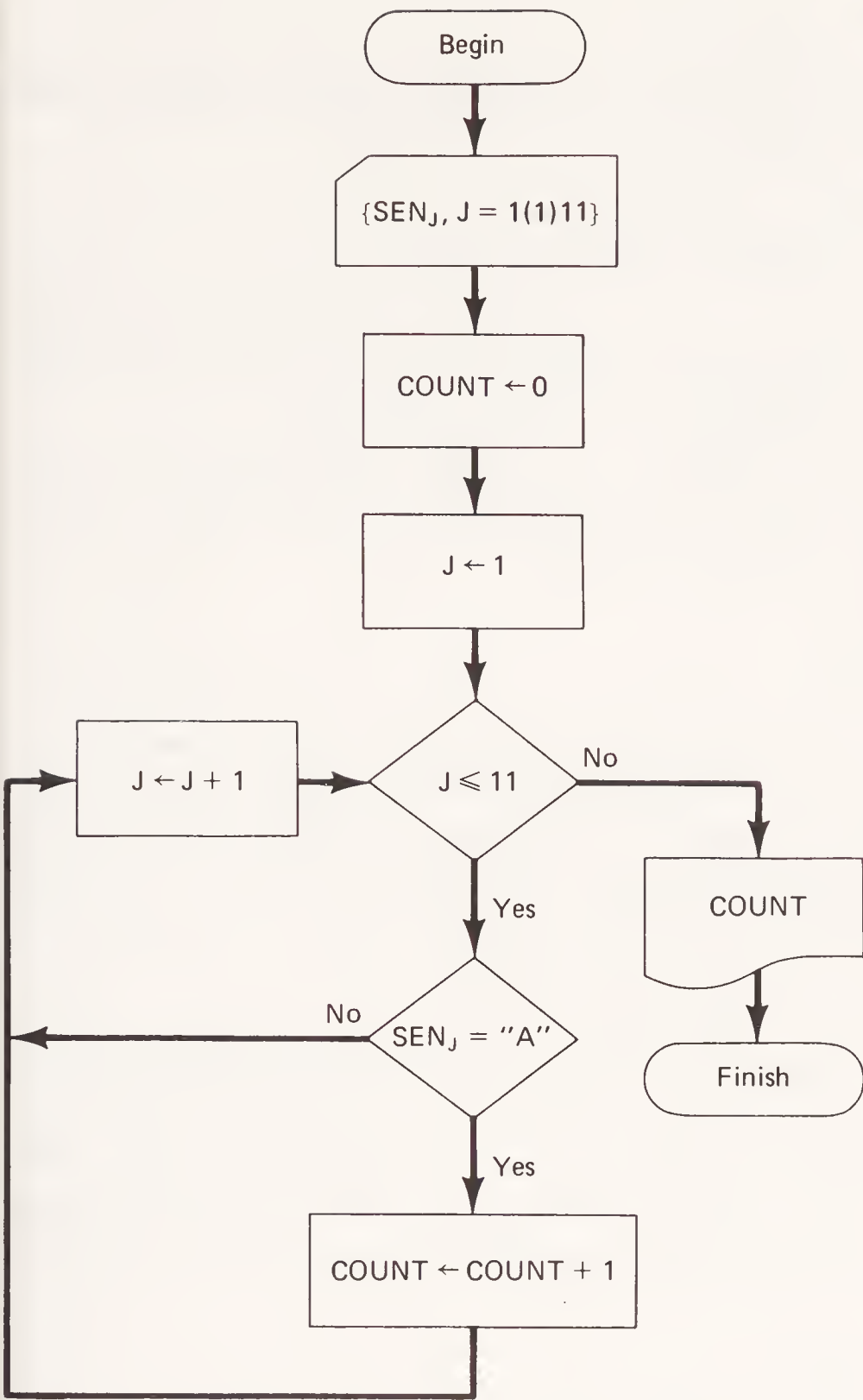


Figure 4-15 Flowchart to count the number of A's in a sentence

“I AM GOING TO TOWN”

Using two vertical bars (||) to denote concatenation, we might represent this process as follows:

$$\text{STR} \leftarrow \text{STR1} \parallel \text{STR2}$$

The right-hand side of this statement is an expression involving character operations.

Another operation applicable to character data is the extraction of a substring from a longer string. This operation is readily implemented in the form of a function analogous to a numerical function such as square root. We might define this function as follows:

SUBSTR (*string*, *p*, *n*)

This function extracts from *string* the *n* consecutive characters beginning with character *p*. For example, the statements

```
SENTENCE ← "DAD IS SAD."
SUBS ← SUBSTR(SENTENCE, 5, 2)
```

gives the value "IS" for SUBS. In a similar fashion, we could define functions to insert substrings, delete substrings from longer strings, and locate substrings within a longer string (often designated as the *index operation*).

At this point, we should probably more clearly distinguish between true numeric data and character data consisting exclusively of numeric digits. That is, the following two statements are not equivalent:

```
X ← 2176
X ← "2176"
```

In the first example, the true numerical value is stored in variable X, and variable X could subsequently appear in arithmetic expressions involving addition, subtraction, etc., but could not appear in a character expression. In the second example, four characters are stored in the storage location reserved for variable X. In this case, variable X could not subsequently appear in an arithmetic expression but could appear in a character expression.

In the processing of input and output statements, the computer utilizes routines that will convert character data consisting solely of numerical digits to numeric data and vice versa. Figure 4-16 illustrates a flowchart to convert the value stored in variable VAL to character data, the result being stored in the elements of array NUM. This flowchart assumes that VAL is a positive integer number. In making the conversion, the array DIG is assumed to be defined as follows:

```
DIG0 ← "0"
DIG1 ← "1"
.      .
.      .
.      .
DIG9 ← "9"
```

More complex cases are considered in the exercises. A similar application also considered in the exercises is the preparation of plots on either a teletype or line printer.

As our final example of the processing of nonnumeric data, let's consider the preparation of a cryptogram. The process begins with an original message, called a *clear*, such as

HE HAD A BAD DAY

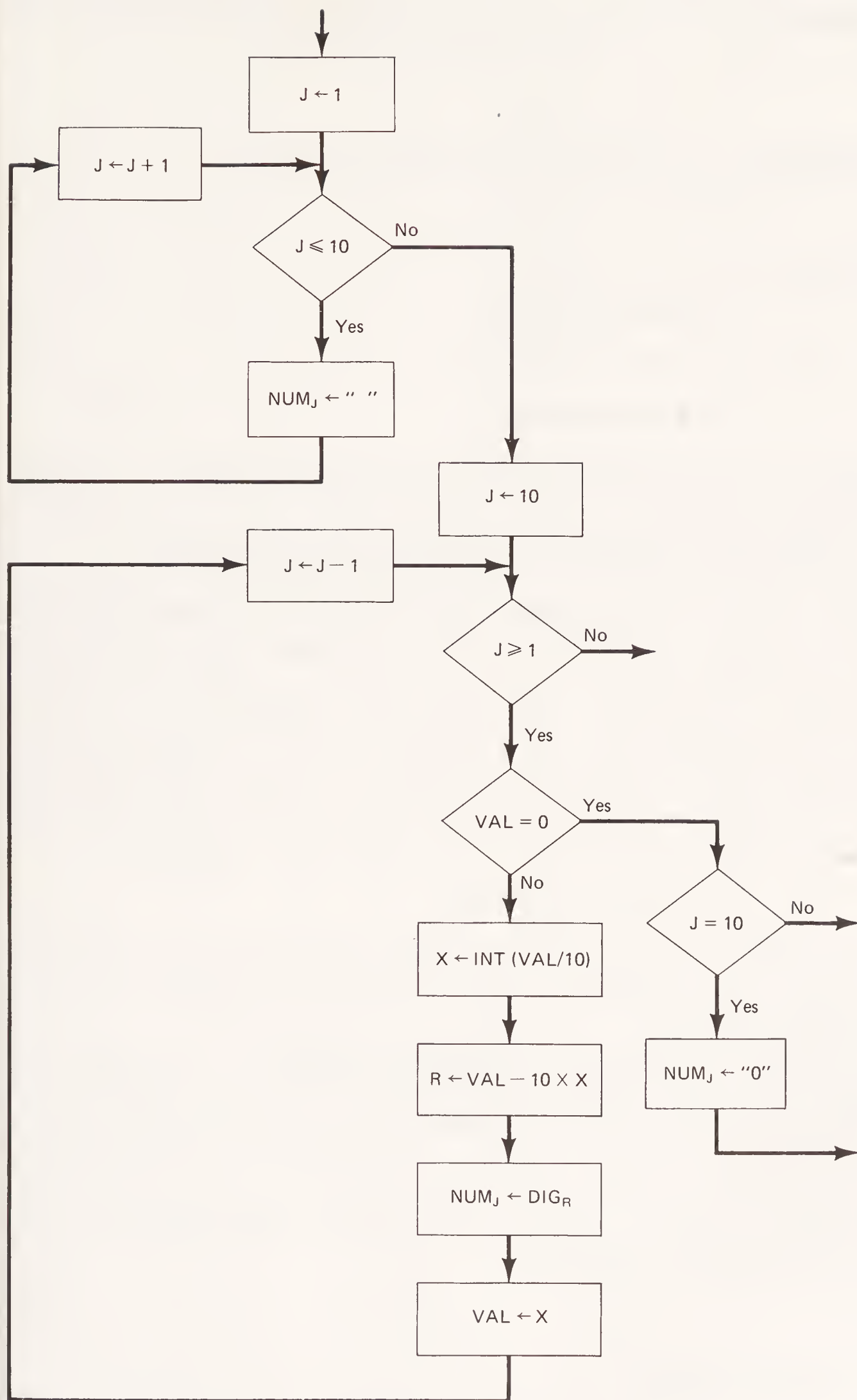


Figure 4-16 Conversion of numeric data to character data

150 consisting of 16 characters. A simple but not very effective approach is to interchange letters. For example, suppose we interchange the letters E and D to obtain

HD HAE A BAE EAY

Interchanging the A with the blank or space character gives

HDAH EA AB EAE Y

Interchanging the H and the Y gives

YDAY EA AB EAE H

Finally, substituting an I for the blanks gives

YDAYIEAIEABIEAEIH

While this cryptogram probably appears completely unintelligible, considerably more effective approaches are available. Converting a clear into a cryptogram is referred to as *enciphering*, while *deciphering* is the reverse operation.

A more effective approach is to use a keyword in the enciphering process, as illustrated in Table 4-4. The first step is to write the message to be enciphered, which is the third line in Table 4-4a. Above the message is written the keyword, repeated as necessary without any blank spaces (line 2 in Table 4-4a). Each of the characters is replaced by a numerical value according to the index table in Table 4-4b, which gives lines 1 and 4 in Table 4-4a. Line 5 is simply the sum of lines 1 and 4. Adding 10 to the results gives line 6. From each value in line 6, the number 27 is subtracted until the result is between 0 and 26. These can then be converted to characters according to Table 4-4b to give the cryptogram:

UCWEGOBM □ XCDGPZ

where □ represents the blank character. Without the keyword, this cryptogram is considerably more difficult to decode than the one obtained by merely interchanging characters.

The flowchart of this process is illustrated in Figure 4-17. The keyword and the clear are read and stored in arrays KEY and MSG, respectively. The index from Table 4-4b is then determined for each character in the keyword and the result stored in array NKEY. The final section of the flowchart looks up the index for each character in the clear, adds the index of the appropriate character in the keyword, adds ten, subtracts 27 as necessary, and then looks up the character corresponding to the result.

4.9. LINEAR LISTS

A type of data structure known as a *linear list* finds applications in one form or another in many areas of computer science. Basically, a linear list is an ordered set of *nodes* that are related in essentially the same manner as the elements of

Table 4-4. Example of Enciphering

A. STEPS IN THE PROCESS

Index of																	
keyword	3	15	13	16	21	20	5	18	3	15	13	16	21	20	5	18	
Keyword	C	O	M	P	U	T	E	R	C	O	M	P	U	T	E	R	
Message	H	E	□	H	A	D	□	A	□	B	A	D	□	D	A	Y	
Index of																	
message	8	5	0	2	1	4	0	1	0	2	1	4	0	4	1	25	
Sum	11	20	13	24	22	24	5	19	3	17	14	20	21	24	6	43	
Add 10	21	30	23	34	32	34	15	29	13	27	24	30	31	34	16	53	
Subtract																	
27 where																	
necessary	21	3	23	7	5	7	15	2	13	0	24	3	4	7	16	26	
Final																	
code	U	C	W	G	E	G	0	B	M	□	X	C	D	G	P	Z	

B. INDEX TABLE USED IN THE ENCIPHERING OPERATION

i	SYM _i	i	SYM _i	i	SYM _i
0	□	9	I	18	R
1	A	10	J	19	S
2	B	11	K	20	T
3	C	12	L	21	U
4	D	13	M	22	V
5	E	14	N	23	W
6	F	15	O	24	X
7	G	16	P	25	Y
8	H	17	Q	26	Z

the single-subscripted array discussed earlier in this chapter. The node is much like an element of the single-subscripted array except that the node may consist of a set of the more primitive data types such as character or numeric data. For example, the following linear list may summarize the last five jobs processed on a batch-processing computer system:

	Account no.	Name	CPU time	Pages output
node 1	1202-5001	J. Bleaux	10.245	15
node 2	1301-5022	C. Smith	19.716	27
node 3	1411-5025	P. Murrill	4.112	56
node 4	1105-4013	J. Doe	48.120	9
node 5	1202-5007	S. Lord	9.766	21

Each node consists of four elements: account number, name, CPU time, and pages of output. More than one word of memory is usually required for each node in the list.

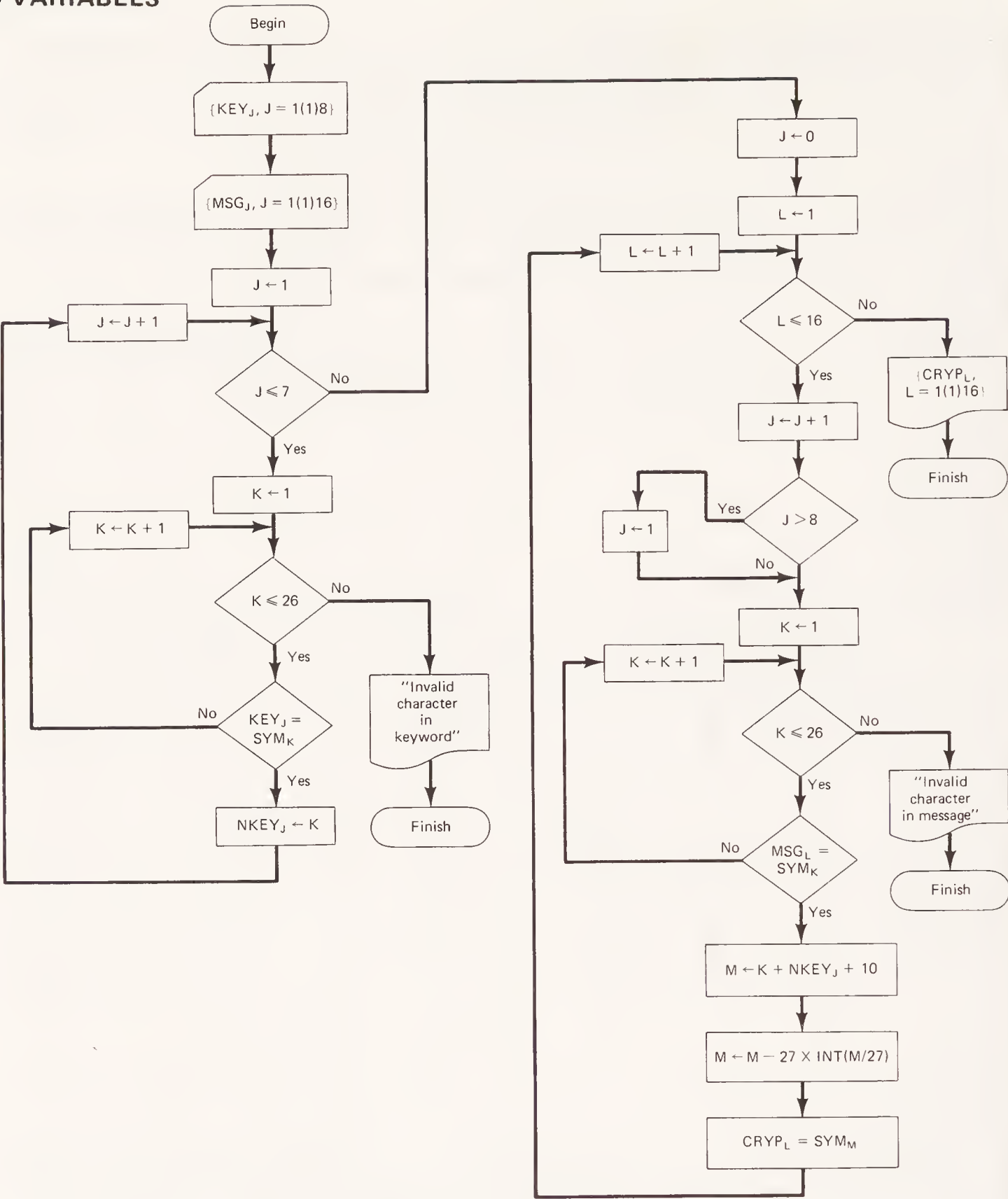


Figure 4-17 Flowchart of the enciphering process of Table 4-4

A variety of operations can be performed on linear lists: any node can be deleted, a node can be added at either end or can be inserted into the list, any node can be accessed for any purpose whatever, the nodes can be rearranged (for example, arranged in ascending order of CPU time used), etc. Furthermore, a linear list can be separated into two linear lists by any appropriate criterion, for example, all who used more than 10 seconds CPU time into one list and all others into another list. Similarly, two linear lists can be combined to form a single list.

In computer science applications, special types of linear lists known as stacks, queues, or dequeues appear most frequently. In effect, all three of these are linear lists with restrictions on the permissible operations as follows:

queue. The only permissible operations are the adding of nodes at one end of the list (the *rear*) and the removal of nodes at the other (the *front*).

deque (double-ended *queue*). The only permissible operations are the addition and/or removal of nodes from either or both ends of the list.

stack. The only permissible operations are the adding and/or removal of nodes from one end (the top) of the list.

A card deck can be used to construct an analogy. When used as a queue, cards are removed from the top and added to the bottom. When used as a deque, they may be added or removed from either end. When used as a stack, cards may only be taken from the top and may only be returned to the top.

Linear lists may be arranged in storage either sequentially or linked. When arranged sequentially, consecutive storage locations are used to store the list. When arranged in a linked fashion, the nodes are not stored sequentially, although the data within each node are stored in sequential memory locations. With each node is stored one or more pointers that specify the first storage location used for the next and/or the prior node in the linear list.

The next section discusses a linked list in more detail. Chapter 9 discusses an application of stacks to compiling.

4.10. ONE-WAY LISTS

In this section, we shall use text editing as another example of a nonnumerical application using a linked list. Specifically, suppose that we begin with the following sentence:

AN APPLE A DAY KEEPS THE DOCTOR AWAY.

Suppose we would like to edit this sentence to remove all the *a*'s, *an*'s, and *the*'s. Doing this, the sentence becomes

APPLE DAY KEEPS DOCTOR AWAY.

Furthermore, suppose we store this sentence in memory utilizing a separate storage location for each character. If the first available storage is at location 1, the original sentence would be stored as in Table 4-5a. The removal of the words one at a time is illustrated in Table 4-5b-d. The drawback of this approach is that it requires a significant amount of shifting characters around in storage. For long texts, this could be very time consuming.

A more efficient manner to arrange storage for editing is to use a list in which each element consists of

1. The character itself.
2. The location of the next character in the list.

The storage of our sentence in a list of this type is illustrated in Table 4-6a. For example, storage location 13 contains the following information:

Table 4-5. Example of Text Editing

a. Original		b. "An" removed		c. "A" removed		d. "The" removed	
Location	Character	Location	Character	Location	Character	Location	Character
1	A	1	A	1	A	1	A
2	N	2	P	2	P	2	P
3	□	3	P	3	P	3	P
4	A	4	L	4	L	4	L
5	P	5	E	5	E	5	E
6	P	6	□	5	□	6	□
7	L	7	A	7	D	7	D
8	E	8	□	8	A	8	A
9	□	9	D	9	Y	9	Y
10	A	10	A	10	□	10	□
11	□	11	Y	11	K	11	K
12	D	12	□	12	E	12	E
13	A	13	K	13	E	13	E
14	Y	14	E	14	P	14	P
15	□	15	E	15	S	15	S
16	K	16	P	16	□	16	□
17	E	17	S	17	T	17	D
18	E	18	□	18	H	18	O
19	P	19	T	19	E	19	C
20	S	20	H	20	□	20	T
21	□	21	E	21	D	21	O
22	T	22	□	22	O	22	R
23	H	23	D	23	C	23	□
24	E	24	O	24	T	24	A
25	□	25	C	25	O	25	W
26	D	26	T	26	R	26	A
27	O	27	O	27	□	27	Y
28	C	28	R	28	A	28	.
29	T	29	□	29	W		
30	O	30	A	30	A		
31	R	31	W	31	Y		
32	□	32	A	32	.		
33	A	33	Y				
34	W	34	.				
35	A						
36	Y						
37	.						

The interpretation is that the character A is stored in this location and that the next character in the text is stored in location 14. In our lists, a “next character in list” indicator of 0 as in location 37 designates the end of the string.

Lists such as in Table 4-6 are referred to as “one-way” lists, since it is possible to proceed only from the beginning of the text to the end. That is, one cannot

Table 4-6. Use of a List

a. Original list		b. Words removed		c. "Unused" list formed	
Location	String element	Location	String element	Location	String element
1	A - 2	1	A - 2	1	A - 2
2	N - 3	2	N - 3	2	N - 3
3	□ - 4	3	□ - 4	3	□ - 9
4	A - 5	4	A - 5	4	A - 5
5	P - 6	5	P - 6	5	P - 6
6	P - 7	6	P - 7	6	P - 7
7	L - 8	7	L - 8	7	L - 8
8	E - 9	8	E - 11	8	E - 11
9	□ - 10	9	□ - 10	9	□ - 10
10	A - 11	10	A - 11	10	A - 21
11	□ - 12	11	□ - 12	11	□ - 12
12	D - 13	12	D - 13	12	D - 13
13	A - 14	13	A - 14	13	A - 14
14	Y - 15	14	Y - 15	14	Y - 15
15	□ - 16	15	□ - 16	15	□ - 16
16	K - 17	16	K - 17	16	K - 17
17	E - 18	17	E - 18	17	E - 18
18	E - 19	18	E - 19	18	E - 19
19	P - 20	19	P - 20	19	P - 20
20	S - 21	20	S - 25	20	S - 25
21	□ - 22	21	□ - 22	21	□ - 22
22	T - 23	22	T - 23	22	T - 23
23	H - 24	23	H - 24	23	H - 24
24	E - 25	24	E - 25	24	E - 0
25	□ - 26	25	□ - 26	25	□ - 26
26	D - 27	26	D - 27	26	D - 27
27	O - 28	27	O - 28	27	O - 28
28	C - 29	28	C - 29	28	C - 29
29	T - 30	29	T - 30	29	T - 30
30	O - 31	30	O - 31	30	O - 31
31	R - 32	31	R - 32	31	R - 32
32	□ - 33	32	□ - 33	32	□ - 33
33	A - 34	33	A - 34	33	A - 34
34	W - 35	34	W - 35	34	W - 35
35	A - 36	35	A - 36	35	A - 36
36	Y - 37	36	Y - 37	36	Y - 37
37	• - 0	37	• - 0	37	• - 0

readily “back up.” Of course, a two-way list could be constructed by storing the address of the previous character in the text in addition to storing the address of the next character in the text.

Editing of this list simply entails changing of the “next character in list”

indicators. For example, removing the word A from our sentence involves changing element 8 of the list from E-9 to E-11, as illustrated in Table 4-6b. Similarly, THE is removed by changing element 20 from S-21 to S-25. Removal of AN from the beginning of the text requires that we note that the new text begins with location 4 instead of location 1. This editing has been achieved without shifting any characters around in storage.

As a result of this editing process, we now have unused locations, that is, locations whose entries are no longer part of the string, interspersed with locations whose contents comprise the string. If the “next character in list” indicators are set properly, these locations can form a separate list of “unused” locations.

Since the AN at the beginning of the list has been removed, the first unused character is in location 1, which becomes the beginning of our “unused” list. The pointers to locations 2 and 3 indicate that these locations are also in our “unused” list. Since the blank in location 9 and the A in location 10 are the next “unused” locations, the pointer in location 3 is changed to 9. Similarly, the pointer in location 10 is changed to 21, the first location in □ THE. Since location 24 is the last location in the unused list, its pointer is set equal to zero.

Now suppose we would like to insert the word SHOULD between the words DAY and KEEP, whose S is dropped, in our edited list, thereby obtaining

APPLE DAY SHOULD KEEP DOCTOR AWAY.

The elements in the unused list can be used to insert SHOULD into the text, giving the result in Table 4-7. Locations 1, 2, 3, 9, 10, 21, and 22 from the unused list have been used to insert the characters □SHOULD. The text still starts with location 4, but the unused list now starts with location 23. The character S in location 20 has been added at the end of the unused list, which necessitated changing the pointer in location 24 from 0 to 20.

To synthesize the use of such lists in a language such as Fortran, we can use two single-subscripted variables, say, C and L, to store the characters and the “next location” pointer. To illustrate their use, suppose these arrays have already been initialized with the original text as in Table 4-6a. The location of the first character in the text is stored in variable ITXT, and initially has the value 1. The location of the first unused location is stored in IFREE, which initially has the value 0, since there are no unused locations.

The flowchart in Figure 4-18 will edit out all of the *a*’s, *an*’s, and *the*’s, updating the indicators in both lists as it proceeds. Variables NTXT and NFREE are used as the current indicators in the two lists, and are initialized to ITXT and IFREE, respectively, in Block 2. Block 3 is entered only at the start of the list and after each blank character in the string. Blocks 3 through 9 determine if the sequence A□ or AN□ occurs. Blocks 10 thru 16 determine if the sequence THE □ occurs. If either of these sequences occur, blocks 17 thru 25 are executed, which update both lists in order to remove the characters. If neither of these sequences occurs, blocks 26, 27 and 28 advance NTXT until the next blank character is located.

Table 4-7 Use of Unused Characters to Insert into the String

Location	Element
1	□ - 2
2	S - 3
3	H - 9
4	A - 5
5	P - 6
6	P - 7
7	L - 8
8	E - 11
9	O - 10
10	U - 21
11	□ - 12
12	D - 13
13	A - 14
14	Y - 1
15	□ - 16
16	K - 17
17	E - 18
18	E - 19
19	P - 25
20	S - 0
21	L - 22
22	D - 15
23	H - 24
24	E - 20
25	□ - 26
26	D - 27
27	O - 28
28	C - 29
29	T - 30
30	O - 31
31	R - 32
32	□ - 33
33	A - 34
34	W - 35
35	A - 36
36	Y - 37
37	• - 0

Location of first element in storage : 4

Location of first element in “unused” list: 23

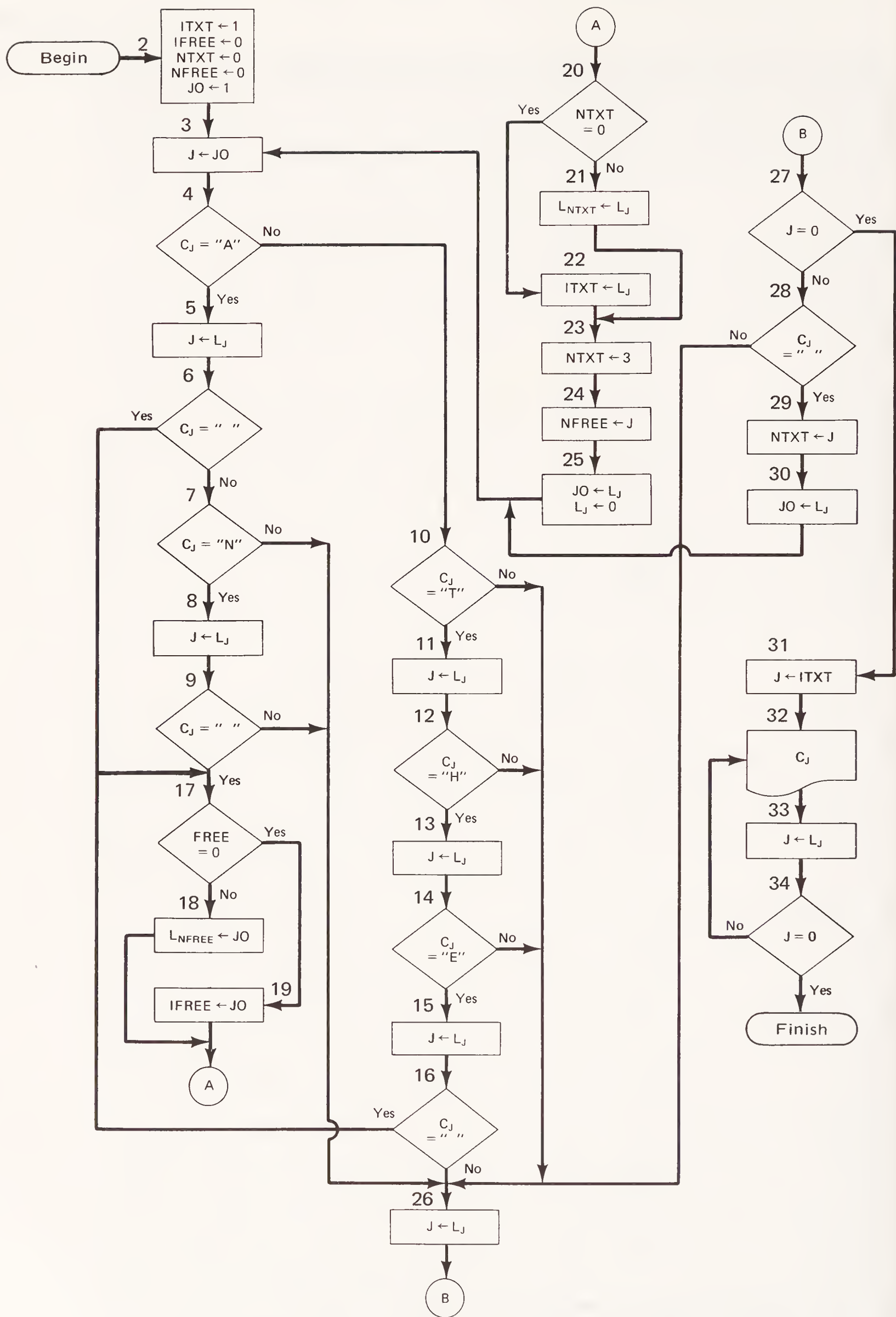


Figure 4-18 Flowchart for editing

EXERCISES

NOTE: As pointed out in the chapter, an understanding of when to use and when not to use subscripted variables is important. To keep you “loose,” a few exercises have been included that do not require the use of subscripted variables.

4-1. Twenty students take an exam on which the marks are from 0 to 100. Prepare the flowchart for a program to determine how many students make a mark higher than the average. The program should write the array of marks and the number earning a mark higher than the average.

4-2. The accounting procedure at the typical computer center consists of assigning a charge number to each user, which he must submit with each job. Suppose at the end of each job, the computer records the user’s charge number and the time of the run in minutes. Prepare the flowchart for a program that reads this data and calculates the total amount of time used by each user on all jobs he ran. The output should be in columnar fashion with user number, total time used, and percent of total. Use a zero entry to indicate end-of-data. Input data are

User no.	Time
40709	1.27
80001	2.34
50200	2.11
40709	4.02
40201	3.11
70207	2.06
50200	3.09
80001	1.02
40709	0.79
70207	3.04

4-3. Consider preparation of a program to calculate telephone bills. Our booming company has ten customers, whose identification numbers and base charges are available from input data. In addition, we have several records containing charges for each long-distance call made along with the customer number of the calling party. Of course, a given customer may make no calls or several calls. The records for long distance are neither counted nor ordered. Prepare the flowchart for a program to compute and print the total charges for each customer. Don’t forget to add 10 percent federal excise tax for long-distance calls, but remember that base charges are not taxable. Enter the base charges first, making up your own data.

4-4. Suppose the sales slips on merchandise sold in a given department contain the amount of the sale and the salesperson’s identification number. We would

like to prepare a program to calculate (1) the total sales of each salesperson, (2) the percent of the total, and (3) his or her commission for the day (3 percent of the sales above \$50). These sales slips are neither counted nor ordered in any way. The program's output should be in columns containing the salesperson's identification number, the total amount of his or her sales for the day, the percent of the total, and his or her commission.

4-5. Modify Exercise 3-19 to calculate the capital-recovery factor for interest rates of 2, 4, 6, 8, and 10 percent for a given year. The output should be six columns, one column for the year and one column for the capital-recovery factor at each interest rate. The year should be read.

4-6. Suppose we are given a set of observations consisting of values of the independent variable x (perhaps the age of a child) and corresponding values of the dependent variable y (perhaps the weight of the child). This data might appear as follows:

x	y
3	31
5	61
2	27
7	75
9	102

Suppose also that it is anticipated that this data should follow an equation of the type $y = ax + b$, where a and b are constants, specifically 10 and 6, respectively, for this case.

A measure of the "goodness" of the equation is to calculate the sum of squares of the differences between observed and predicted (that is, by the equation) values of the dependent variable. For example, for the first observation, the predicted value of y is $10 \times 3 + 6 = 36$, the observed value is 31, giving a difference of 5. The sum of squares of these values for each value of x is the quantity desired. Prepare the flowchart for a program to calculate this value for the above data.

4-7. An array of data consists of n experimental measurements of the variable x . Read them and calculate their mean value and standard deviation:

Mean value $= \frac{1}{n} \sum_{i=1}^n x_i = \bar{x}$

Standard deviation $= \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$

Prepare the flowchart for a program to write the mean value, the standard deviation, and n .

4-8. Redo Exercise 4-7 except in this exercise write the value of x which is most different from the mean value of x .

4-9. A single-subscripted array X contains 20 elements which represent experimental measurements. These may be smoothed by calculating

$$sX_i = \frac{X_{i-1} + X_i + X_{i+1}}{3}$$

for all but the first and twentieth elements. Read X , calculate sX_i and write the two arrays, one above the other.

4-10. Five students take four examinations. Their marks are

	Exam 1	Exam 2	Exam 3	Exam 4
Student 1	48.6	30.	62.8	23.4
Student 2	40.1	40.	60.1	29.6
Student 3	63.4	50.	63.7	31.2
Student 4	56.2	60.	58.2	27.3
Student 5	71.0	70.	67.3	26.4

Read their marks as a double-subscripted array. Calculate the average on each test, the average for each student on all four tests, and the average for all students on all tests. In calculating the averages, Exam 4 is to be given twice as much weight as the other exams.

4-11. Three construction men work a week (five days) and put in the hours shown:

	Day 1	Day 2	Day 3	Day 4	Day 5
Worker 1	8.0	8.5	9.5	8.0	8.5
Worker 2	8.0	8.5	10.0	8.0	9.0
Worker 3	8.0	9.0	9.0	9.0	8.5

Read their hours as a double-subscripted array. Calculate the total hours worked for each man, and the total hours worked by all men in the week.

4-12. One very useful item to a contractor on a construction project is the number of men of each craft required each week. Suppose he will need carpenters, plumbers, and electricians, hereafter referred to as crafts 1, 2, and 3, respectively. The contractor usually divides his total effort into various jobs, which he then schedules to start on a given week. For each job, he estimates how many weeks are required and how many of each craft are needed. On each job information is available containing the job number, the starting week, the weeks required, and the number of workers in crafts 1, 2, and 3.

Assuming the total duration of the contract is 10 weeks, prepare the flow-chart for a program that determines the number of workers in each craft needed each week. The output should be in columnar fashion, with the week-number and the number of craftsmen needed in each craft. Use as input the following:

Job	Start	Duration	Craft 1	Craft 2	Craft 3
01	2	4	5	1	2
02	1	2	2	0	0
03	4	1	0	4	0
04	2	2	3	0	1
05	5	5	2	0	0
06	7	1	0	1	0
07	9	1	1	1	1

4-13. Read a single-subscripted array A containing eight elements. Sort the elements into an ascending order based on their absolute value, and write the results.

4-14. Let $f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$. A program is to be prepared to read n , read the coefficients of $f(x)$, read b and evaluate $f(b)$. The first entry is the value of n , the next $n + 1$ entries are the values of the coefficients (beginning with a_0), and the last entry is the value of b . Store the coefficients of $f(x)$ in a single-subscripted array. Print b and $f(b)$.

As input, let $f(x) = -0.8x^5 - 0.06x^4 + 1.7x^3 - 3.2x^2 + 7x + 1$ and evaluate at $x = 0.75$. Although not the easiest to program, the most efficient manner to evaluate this polynomial is as follows:

$$(((((-0.8x - 0.06)x + 1.7)x - 3.2)x + 7)x + 1$$

4-15. Let $f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$, let $g(x) = cx + d$, and let $h(x) = f(x)g(x) = b_{n+1}x^{n+1} + b_nx^n + \cdots + b_1x + b_0$. Prepare the flowchart for a program to read n , c , and d , read the coefficients of $f(x)$ (beginning with a_0), calculate the coefficients of $h(x)$, and print the results. Use array A to store the coefficients of $f(x)$ and similarly for the coefficients of $h(x)$.

As input, let $g(x) = 1.7x + 2.1$ and let $f(x) = x^5 + 1.6x^4 - 0.7x^3 + 0.2x^2 + 6.1x + 0.8$.

4-16. As a conscientious student in basket weaving, Joe Kollage has performed some bursting tests on his products. Since higher mathematics still baffle him, he would like you to write a program for him.

The experiment consisted of placing weights in each basket until it ruptured. Joe used three different weights—weight 1 weighing 5 pounds, weight 2, 10 pounds, and weight 3, 25 pounds. He performed his experiment on four baskets, and he has the number of weights of each type in the basket when it ruptured. The data for each test are as follows:

	Number of weight 1	Number of weight 2	Number of weight 3
Test 1	12	1	0
Test 2	6	2	1
Test 3	8	1	1
Test 4	0	2	2

Joe is confident he can keep the entries in order, so only the number of weights of each type for each test is entered. The program should proceed as follows:

1. Read input.
2. Calculate total weight in basket at rupture. This value should be stored in an array.
3. Calculate average weight at rupture.
4. Calculate the difference between individual test value and average test value.
5. Print test number, total weight at rupture, and difference from average for each test.
6. Print average weight at rupture.

This can be accomplished with only two loops.

4-17. Joe Bleaux needs to borrow \$1,500 for a period of three years. After consulting three loan departments, he has the following possibilities:

1. 8 percent per year, compounded monthly
2. 8 1/2 percent per year, compounded annually
3. 8 1/4 percent per year, compounded quarterly

Which one is the most attractive?

Prepare the flowchart for a program to calculate this. Read i , the annual interest rate, and m , the number of compounding periods annually. The final amount owed is $\$1,500 \cdot (1 + i/m)^{3m}$. The output should be the annual interest rate, the number of compounding periods annually, and the amount owed after three years. The output should be ordered so that the most attractive appears first and the least attractive last. Assume that the data are *not* read in this order.

4-18. Suppose we have the coordinates of five points in the x,y plane, and would like to find the distance between the two points that lie the farthest apart. The five points are

x	y
-0.94	-3.22
-4.02	8.17
7.07	-9.11
5.49	8.76
0.20	4.45

164

The distance between the first two points is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, between the first and third is $\sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}$, etc. Only the value of the farthest distance is to be printed. Be sure to check all possibilities (a total of 10).

4-19. Suppose we have an item of equipment worth \$1,000 which we would like to depreciate over a five-year lifetime. We propose to do this via one of two ways:

- 1. Double-declining-balance method, where the depreciation for the year is $2/n$ times the book value at the beginning of the year (n = number of years over which item is to be depreciated).
- 2. Straight-line method, where the depreciation is $1/n$ times the original value of the equipment.

Prepare the flowchart for a computer program to calculate the depreciation by each of these methods and print a table, similar to the following:

Year	Double-declining-balance depreciation	Beginning book value	Straight-line depreciation	Beginning book value
1	\$400.00	\$1,000.00	\$200.00	\$1,000.00
2	\$240.00	\$ 600.00	\$200.00	\$ 800.00
3	\$144.00	\$ 360.00	\$200.00	\$ 600.00
4	\$ 86.40	\$ 216.00	\$200.00	\$ 400.00
5	\$ 51.84	\$ 129.60	\$200.00	\$ 200.00

4-20. As can be seen from the table accompanying the previous problem, the double-declining-balance depreciates equipment at a faster initial rate than does the straight-line method (\$400 versus \$200, or twice as fast). This method will never reduce the book value of zero, however, since the depreciation is always a fraction of the present book value. This can be circumvented by using the double-declining-balance method during the initial years, and switching to the straight-line method at a later date. In fact, the switch should be made whenever the depreciation via the straight-line method would exceed that from the double-declining-balance method.

Prepare the flowchart for a program that depreciates a given amount over a given number of years (both read as inputs) via the above method. The output should appear as follows for depreciating \$1,000 over a five-year period:

Year	Depreciation	Beginning book value
1	\$400.00	\$1,000.00
2	\$240.00	\$ 800.00
3	\$144.00	\$ 360.00
4 ^a	\$108.00	\$ 216.00
5	\$108.00	\$ 108.00

^aSwitched to straight-line method.

4-21. Let $f(x)$ be defined by the following equation:

$$f(x) = x^2 \sin (\pi x)$$

Prepare the flowchart for a program to perform the following functions:

- 1. Compute $f(x)$ at values of x at 0, 0.1, 0.2, . . . , 1.0.
- 2. Locate the maximum of these values.
- 3. Divide each of the above values by the maximum value to obtain normalized values.
- 4. Print as shown in the following table.

x	$f(x)$	$f(x)$
0	0	0
0.1	0.00031	0.00078
0.2	0.02351	0.05931
0.3	0.07281	0.18367
0.4	0.15217	0.38386
0.5	0.25000	0.63065
0.6	0.34238	0.86369
0.7	0.39642	1.00000
0.8	0.37618	0.94895
0.9	0.2503	0.6314
1	0	0

4-22. Financial analysts take into account the time value of money via the present value. To illustrate this concept, suppose an investment is expected to yield the returns given in the second column of the following table:

Year	Return	Present value
1	\$ 7,000	$7,000/1.08 = 6,481$
2	9,000	$9,000/(1.08)^2 = 7,716$
3	13,000	$13,000/(1.08)^3 = 10,320$
4	20,000	$20,000/(1.08)^4 = 14,701$
5	40,000	$40,000/(1.08)^5 = 27,233$
		Present value = 66,441

The present value of the return is the amount that would have to be invested now to yield this return, as illustrated in the above table for an interest rate of 8 percent.

Prepare the flowchart for a program to calculate the present value of the above investment for interest rates of 8, 12, and 15 percent. Store the returns in an array.

4-23. Modify the program for Exercise 4-2 so that the output contains the

166 user's name along with his charge number. Assume this particular center has only ten users, whose names and corresponding charge numbers are as follows:

John Anderson	70206
Frank Wilson	40709
Joe East	50200
Myron Johnson	40201
Jack Evers	40710
Henry Williams	70207
John Smith	80001
Michael Sand	50201
Ralph Blue	70208
Horace White	40711

Using the time cards given as data for Exercise 4-2, the output of the program should appear as follows:

User	Charge no.	Time used	Percent
Frank Wilson	40709	6.08	26.6033
Joe East	50200	5.2	22.7571
Myron Johnson	40201	3.11	13.6105
Henry Williams	70207	5.1	22.3195
John Smith	80001	3.36	14.7046

Notice that the user's name is not printed if he did not use any time.

4-24. Prepare the flowchart for a program to perform the following:

1. Read the number of students in a class.
2. Read one quiz score for each student along with his name.
3. Arrange the scores in descending order.
4. Compute the average score.
5. Write average grade, the number of students making above average, and the number of students making below average.
6. List the scores in descending order along with names. Write "Median" by median grade.

An example is shown in the table below.

AVERAGE SCORE	78.2857
NUMBER ABOVE AVERAGE	4
NUMBER BELOW AVERAGE	3
SCORE	NAME
99	ROBERTS
91	STEPHENS

SCORE	NAME	
87	JONES	
82	KILROY	MEDIAN
71	LEVEQUE	
62	NO NAME	
56	JAMES	

4-25. The following timetable is given for train schedules:

Distance	City	Time, PM
0	Chicago	4:00
93	Niles	6:42
141	Kalamazoo	7:28
164	Battle Creek	7:53
210	Jackson	8:39
248	Ann Arbor	9:18
284	Detroit	10:00

Prepare a program to read the above information, calculate the remaining entries in the table below, and print in a similar fashion.

TRAIN SCHEDULE

DISTANCE	CITY	TIME PM	TIME FROM LAST STOP, MIN	ELAPSED TIME, MIN
0	CHICAGO	4 00	0	0
93	NILES	6 42	162	162
141	KALAMAZOO	7 28	46	208
164	BATTLE CREEK	7 53	25	233
210	JACKSON	8 39	46	279
248	ANN ARBOR	9 18	39	318
284	DETROIT	10 00	42	360

4-26. Consider writing a program to update a student's grade-point average. Suppose data on each student are available as follows:

- 1. Student's name
- 2. Number of credit hours taken prior to this semester, followed by the number of credit hours earned.
- 3. Number of quality points earned prior to this semester.
- 4. The number of hours for his first class followed by the grade.
- 5. The number of hours for his second class followed by the grade.
- 6. etc. End of data is indicated by a negative course hour.

168 A typical set of input might appear as follows:

JOE KOLLAGE	65	62	181	3B	3C	1A	4B	5B
KILROY	71	65	145	3C	5D	3F	3C	
JOE BLEAUX	48	48	159	3A	3B	5A	4B	

Grade-point ratios are computed on the 4-point system, that is, 4 points for A, 3 points for B, 2 points for C, 1 point for D (hours are earned), and 0 points for F (hours are not earned).

Prepare the flowchart for a computer program that reads these data and prints the student's name, the number of credit hours taken, the number of credit hours earned, and his quality point ratio at the end of the term. If his quality point ratio is less than 2.0, place an asterisk prior to his name, as follows for the above input:

NAME	HOURS TAKEN	HOURS EARNED	POINT RATIO
JOE KOLLAGE	81	78	2.80247
* KILROY	85	76	1.90588
JOE BLEAUX	63	63	3.36508

4-27. Consider writing a program to print bank statements. The first entries for each account are the customer's name (15 characters), the second line of his address (15 characters), the third line (15 characters), his account number, and the balance at the beginning of the month. Each succeeding entry is a transaction, consisting of the date (day, month, and year) and the amount of the transaction (negative transactions indicate a check). A zero transaction indicates the end of the data. Prepare the flowchart for a program to process these data and print a statement as shown in the following figure. Note that if a check and a deposit are made on the same day, they are written on the same line. The service charge is computed as follows:

- 1. No charge if the minimum balance is \$400.00 or more.
- 2. A charge of 60 cents plus 4 cents per check less 20 cents per \$100.00 of the minimum balance when under \$400.00.

THE PHANTOM
2022 DUNGEON WAY
OUT OF THIS WORLD

ACCOUNT NUMBER 1107

DATE	WITHDRAWAL	DEPOSIT	BALANCE
			412.41
3 / 7 / 44	7.07		405.34

DATE	WITHDRAWAL	DEPOSIT	BALANCE	169
3 / 9 / 44	12.41		392.93	
3 / 9 / 44	45.12		347.81	
3 / 15 / 44	57.21		290.60	
3 / 19 / 44	107.49	32.52	215.63	
3 / 22 / 44		12.12	227.75	
3 / 22 / 44		47.19	274.94	
3 / 27 / 44	52.12		222.82	
4 / 5 / 44		202.17	424.99	
	0.47 SC		424.52	

4-28. The subscripted variable feature can be used to construct a more compact flowchart for Exercise 2-33. Specifically, use an array to store the names of the months and a second array to store the maximum number of days. Now only February must be handled in a special manner.

4-29. Exercise 2-37 treated the computation of the bill for a car rental. Instead of entering the total number of hours the car has been kept, the time and date at which it was rented are entered followed by the time and date that it was returned. For example, an entry might be as follows:

4:54P 8-18-72

The program should then compute the number of hours the car has been rented. For this purpose, the program must use an array containing the number of days in each month as in the previous exercise. The program should also function properly if the car is kept over New Year. Don't worry about leap years unless you are especially ambitious.

4-30. Another modification to Exercise 2-37 would be to keep a table containing the rates on each type of car rented by our company. For each type of car, this table would contain the daily rate and the cents-per-mile charge. Now the clerk only has to enter the type of car, that is, type 1, type 2, etc. A maximum of six types will be used, so the program should check for unrealistic entries.

4-31. On large turnpike systems such as the New Jersey Turnpike, automobiles may enter and leave at varying points. Suppose we construct a simple system to audit the collections made by the toll collector at one of the exits. Upon entry to the turnpike system, each motorist is given a card on which his point of entry is punched. Upon exit from the turnpike system, he surrenders this card to the toll collector and pays the toll. At the end of the day, the toll collector counts his money and submits this along with the cards to the accounting department. They then enter the amount collected and the gate at which it was collected on a card. This card followed by the cards on which the entry gate was punched are then processed. Using the table of tolls in the accompanying figure, the amount of fare that should have been collected from each motorist can be computed.

These can be summed to give the total amount that should been reported by the toll collector. The program should print this amount along with the amount reported by the toll collector:

Entry gate	EXIT GATE					
	1	2	3	4	5	6
1	—	0.45	1.20	1.85	2.45	3.00
2	0.45	—	0.80	1.50	2.15	2.65
3	1.20	0.80	—	0.80	1.50	2.00
4	1.85	1.50	0.80	—	0.75	1.25
5	2.50	2.20	1.50	0.75	—	0.60
6	3.10	2.75	2.00	1.25	0.60	—

4-32. The flowchart in Figure 4-16 assumes that the number is a positive integer number. Modify this flowchart so that it applies to positive or negative numbers. In effect, array NUM in Figure 4-16 contains the characters as they would appear in the printed number. For this problem, specify the characters in NUM so that positive numbers are written without sign, but negative numbers are written with a minus sign preceding the leftmost digit.

4-33. Extend the flowchart in the previous exercise to non-integer numbers, where *N* digits are to appear to the right of the decimal point. The decimal point should also be stored in the proper character of array NUM. Round the rightmost digit. Let NUM have a maximum of 20 elements.

4-34. Modify the flowchart in the previous exercise so that the number is written in normalized exponential format. That is, the number will be expressed as a fraction between 0.1 and 1.0 followed by an integer exponent. For example, the number 0.027427 would be expressed to four significant figures as follows:

0.2743E-01

Two digits are always used for the exponent. Positive numbers and exponents appear without sign. Let the number of significant figures in the number be *N*, and always round the rightmost digit.

4-35. Modify Exercise 3-31 so that the digits in the binary number are stored in an array. Reserve 32 elements for the array, and set all leading digits to zero. Assume that the number is positive.

4-36. Modify Exercise 3-33 so that the digits in the binary number are stored in an array. Store only the digits in the array. The decimal fraction will always be expressed to 24 binary digits.

4-37. Write a program that reads a sentence that contains a maximum of 80 characters (including the period at the end) and counts the total number of characters (excluding blanks and commas) in the sentence. The characters should

be read into an array as one character per element. Only alphabetic characters, blanks, and commas will appear in the sentence. A comma is always followed by a blank.

4-38. Same as Exercise 4-37 except count the number of occurrences of the letter E in the sentence.

4-39. Same as Exercise 4-38 except count the number of occurrences of each letter of the alphabet within the sentence. The output should appear as in the accompanying figure.

ORIGINAL SENTENCE			
AN IMPORTANT CHARACTERISTIC OF DIGITAL COMPUTERS IS THEIR INCREDIBLE SPEED.			
CHAR.	OCCURRENCES	CHAR.	OCCURRENCES
A	5	N	3
B	1	O	3
C	5	P	3
D	3	Q	0
E	7	R	6
F	1	S	4
G	1	T	7
H	2	U	1
I	9	V	0
J	0	W	0
K	0	X	0
L	2	Y	0
M	2	Z	0

4-40. Same as Exercise 4-37 except count the number of occurrences of words containing three letters or less.

4-41. Same as Exercise 4-40 except compute the average length of the words in the sentence.

4-42. Instead of processing a single sentence, suppose we process a paragraph. The paragraph will contain a maximum of 10 lines as illustrated in the accompanying figure. Each line will contain a maximum of 80 characters. A blank character always follows a period, and no hyphenated words will be used. For this exercise, the paragraph should be read into a single array of 800 elements, one character to the element. Compute the average number of words in the sentences in the paragraph.

4-43. Same as Exercise 4-42 except print the number of words in each sentence and the average number of characters per word for each sentence.

WORD LENGTH DISTRIBUTION	
CHS./WORD	OCCURRENCES
1	5
2	16
3	15
4	6
5	8
6	8
7	12
8	3
9	7
10	2
11	1
12 OR MORE	1

Distribution for Exercise 4-44

- 4-44. Same as Exercise 4-42 except print the distribution of the number of characters in each word as shown in the figure on page 171. Words containing over 12 characters can be considered as containing 12 characters.
- 4-45. Same as Exercise 4-42 except process the lines one at a time. That is, the array should be defined as containing only 80 characters. A line is read into this area, the computations performed, and then the next line is read into this same area.
- 4-46. Same as Exercise 4-43 but with the modification suggested in Exercise 4-45.
- 4-47. Same as Exercise 4-44 but with the modification suggested in Exercise 4-45.
- 4-48. Same as Exercise 4-45 except that the last word on a line may be hyphenated (using the minus sign) and split onto two lines.
- 4-49. Same as Exercise 4-46 except that the last word on a line may be hyphenated and split onto two lines.
- 4-50. Same as Exercise 4-47 except that the last word on a line may be hyphenated and split onto two lines.
- 4-51. Prepare a flowchart for a program to read a string of 80 characters containing an integer number, storing the characters in an array, one character per element. The number may contain a maximum of 12 digits, and may be preceded by a minus sign, by a plus sign, or no sign at all. The number may be preceded by any number of blanks but will not contain any embedded blanks. This number is to be stored in a simple variable.
- 4-52. Same as Exercise 4-51 except that the number may have a fractional part. As before, the number should be read as character data and then converted to numerical data. The number may contain only the decimal digits, a sign (optional), and a decimal point. The following examples should be acceptable: 127.23, + 12.72, - 1200., + .02007; or .00030.
- 4-53. Same as Exercise 4-51 except that the number should be read as character data in the exponential format described in Exercise 4-34. The number may contain only the decimal digits, an optional sign for the exponent, an optional sign for the fraction, the letter E, and a decimal point (in the fraction only).
- 4-54. Write a program to decode the cryptogram produced by the program in Figure 4-17 and obtain the original message.

4-55. Prepare the flowchart for a program to produce a cryptogram via the interchanging of characters as described at top of page 150. The program should read the clear (maximum of 80 characters), followed by pairs of characters to be interchanged. Let an asterisk indicate end-of-data. The output should appear as in the accompanying figure.

ORIGINAL MESSAGE
AN IMPORTANT CHARACTERISTIC OF DIGITAL COMPUTERS IS THEIR INCREDIBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS I AND X
AN XMPORTANT CHARACTERXSTXC OF DXGXTAL COMPUTERS XS THEXR XNCREDXBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS A AND N
NA XMPORTNAT CHNRNCTERXSTXC OF DXGXTNL COMPUTERS XS THEXR XACREDXBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS G AND V
NA XMPORTNAT CHNRNCTERXSTXC OF DXVXTNL COMPUTERS XS THEXR XACREDXBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS M AND K
NA XKPORTNAT CHNRNCTERXSTXC OF DXVXTNL COKPUTERS XS THEXR XACREDXBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS C AND U
NA XKPORTNAT UHNRNUTERXSTXU OF DXVXTNL UOKPCTERS XS THEXR XAURED XBLE SPEED.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS D AND P
NA XKDORTNAT UHNRNUTERXSTXU OF PXVXTNL UOKDCTERS XS THEXR XAUREPXBLE SDEEP.

CRYPTOGRAM AFTER EXCHANGING CHARACTERS A AND G
NG XKDORTNGT UHNRNUTERXSTXU OF PXVXTNL UOKDCTERS XS THEXR XGUREPXBLE SDEEP.

5

FUNCTIONS AND PROCEDURES

The modular approach to the design and construction of physical entities has become extremely popular in recent years. This approach first gained widespread acceptance in the electronics industry. Complex units are easier to design and construct from modules and subsequently are easier to repair, since the repairman can test and replace entire modules. The modular approach has recently been extended to the building trades. For example, a contractor can now buy a complete bathroom or kitchen as a unit, coming partially assembled and with instructions that permit completion of the job with a minimum of effort. The advantages to the contractor are severalfold: (1) he places a single order for the unit as opposed to ordering each item individually; (2) the reduction in on-site labor is substantial; and (3) he no longer has to design the floor plan or piping runs. In fact, the current trend in architecture is to design an entire subdivision of houses by varying combinations of a fixed number of modules. That is, modules are available for the front porch, the living room, dining room, etc.

For algorithmic processes, the concept of a subalgorithm can be extremely useful in developing the total algorithm, especially when the problem is large. We might think of a “master” algorithm that specifies the steps required in achieving the solution to the original problem. This master algorithm may reference one or more subalgorithms that in turn specify the steps required to perform certain operations that must be performed in solving the original problem. These subalgorithms may often be prepared and verified independently, thus conveniently permitting several different individuals to simultaneously work on a large problem. In many cases, the “master” algorithm will reference the same subalgorithm from more than one step. Furthermore, a subalgorithm may reference other subalgorithms or, in some cases, may even reference itself.

In the computer field, the counterpart to the use of a subalgorithm is the use of functions and procedures. For example, suppose a programmer wants to find the largest of three values, say, the values of variables A, B, and C. One approach would be to insert the statements corresponding to the flowchart in Figure 5-1

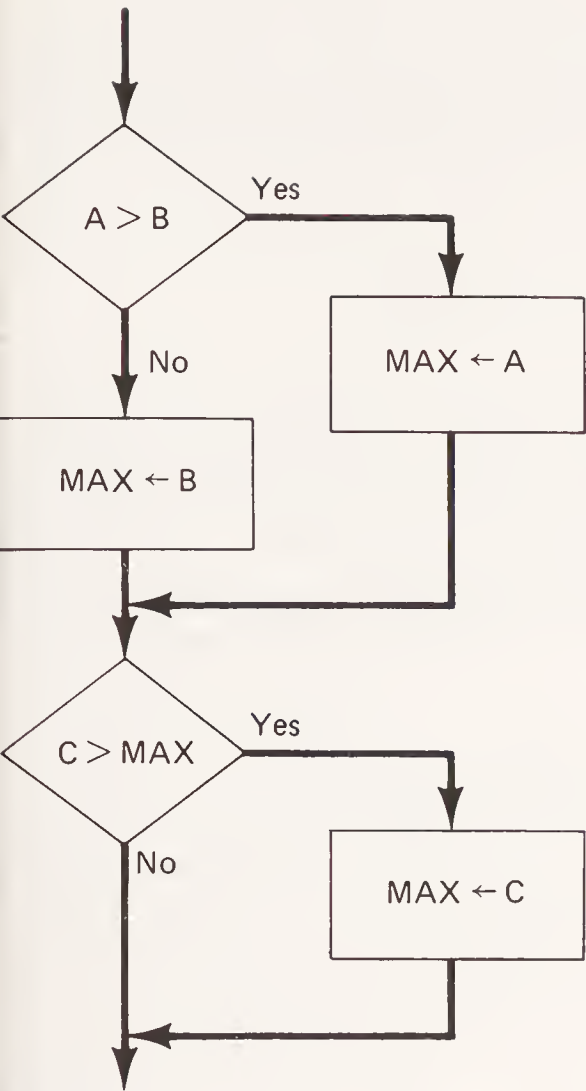


Figure 5-1 Finding the maximum value

at the proper point in his program. An alternate approach is to define a block of code to perform this specific purpose. Each time he wants to find the maximum value of three variables, the programmer simply refers to this block of code. Since he may not always want to find the maximum value of the same three variables, it is important that some means be provided whereby this block of code can be used for different sets of input values.

The function or procedure could be related to the overall program in one of two ways. One approach is to simply insert the block of code in Figure 5-1 into the program each time a reference to it occurs. In order to process different sets of values, the three variables A, B, and C in Figure 5-1 are simply replaced by the three variables whose maximum value is to be found. In programming, functions used in this manner are referred to as *open* functions.

An alternate approach is to use only one block of code, establishing an appropriate *linkage* mechanism whereby the values of the three variables are communicated to the function or procedure, and the result is communicated back to the section of the program from which the reference was made. Functions used in this manner are referred to as *closed* functions. When the same function or procedure is referenced from several different locations within the program, this approach reduces the size of the complete program by not repeating the code. For especially small subalgorithms, however, establishing the linkage to the block of code may approach the size of the block of code itself.

Depending upon the programming language, functions and procedures may be

176 implemented in various ways. Some languages have a block structure that readily permits the implementation of *internal* procedures. In others, the procedure is implemented as a block of code that is compiled separately and referred to as a *subprogram*.

5.1. FUNCTIONS

A functional relationship between two variables exists when, given a value for one variable, a unique value of the second variable is designated by the relationship constituting the function. An example of a functional relationship could be a person's height or his birthday. These functional relationships could be expressed for Joe Kollage as follows:

HEIGHT (J. Kollage) = 6 ft., 4 in.
BIRTHDAY (J. Kollage) = Feb. 27, 1940

In this case, the person's name constitutes the *argument* of the function. In general, these functional relationships become

HEIGHT (X)
BIRTHDAY (X)

In this representation, any person's name can be substituted for the argument X and the function evaluated.

While these two functional relationships can be applied to everyone, many functions are defined only for a restricted set of values of the argument. Suppose we postulate a function TELEPHONE (X) that designates a person's telephone number. The function

TELEPHONE (J. Kollage)

is meaningful only if Joe Kollage has a telephone. Similarly, if we restrict our thinking to real numbers, the square root function, $\sqrt{x}$, is meaningful only for $x \geq 0$.

Another difficulty with the function TELEPHONE (X) is that many people have more than one telephone number. In this case, which one is designated by TELEPHONE (X)? In most cases, this ambiguity could be removed by designating the person's home or residence number, that is,

RESIDENCE TELE (X)

The functional relationship must be such that it clearly designates a single result.

A common case of the requirement of a unique result is in reporting class enrollment. If we simply designate our function as

ENROLLMENT (Basket Weaving 40)

do we mean on the first class day, after the last date for adding courses for the term, at midterm, or at the end of the term? Adoption of a specific day, such as the twelfth day of classes, on which to count heads removes the ambiguity.

Although we have restricted our discussion to functions with only one argument, most programming languages accept functions with two or more arguments. For example, if Basket Weaving 40 is taught each term, then we also need to designate the term. But instead of defining a separate function for each term, we might simply let the term be a second argument.

ENROLLMENT (Basket Weaving 40, Fall)

This function designates the enrollment (on the twelfth day of classes) in Basket Weaving 40 during the Fall term. If our function is to apply over several years, we might let the year be a third argument:

ENROLLMENT (Basket Weaving 40, Fall, 1970)

The value of the function is the enrollment in Basket Weaving 40 for the Fall term, 1970.

While mathematics texts often treat the existence of a function, in this text we shall be concerned only with a subset of these, namely, those that can be computed. For example, restricting our thinking to real numbers, it is well-known that the square-root function $\sqrt{x}$ exists for all $x \geq 0$. But given a value for x , how do you compute the function $\sqrt{x}$?

In the exercises at the end of Chapter 3, the following approach was suggested:

1. Start with $g = 1$
2. Compute

$$h \leftarrow x/g$$

$$g \leftarrow (h + g)/2$$

3. Continue step 2 until

$$|h - g| < \text{TOL}$$

The flowchart of a program to read and compute the square root of x is given in Figure 5-2.

While the flowchart in Figure 5-2 certainly contains the necessary logic for computing square roots, the square-root computation is frequently required in support of another program and not as a program in itself. For example, Figure 5-3 gives the flowchart of an algorithm to compute the roots of the equation:

$$ax^2 + bx + c = 0$$

This flowchart requires the square-root computation in two places: once in block 5 and once in block 7.

In order to utilize the logic in Figure 5-2 to accomplish the necessary square-root computations, we construct what we shall call a reference flowchart, as illustrated in Figure 5-4. In place of "Start" and "Finish" blocks, our terminal

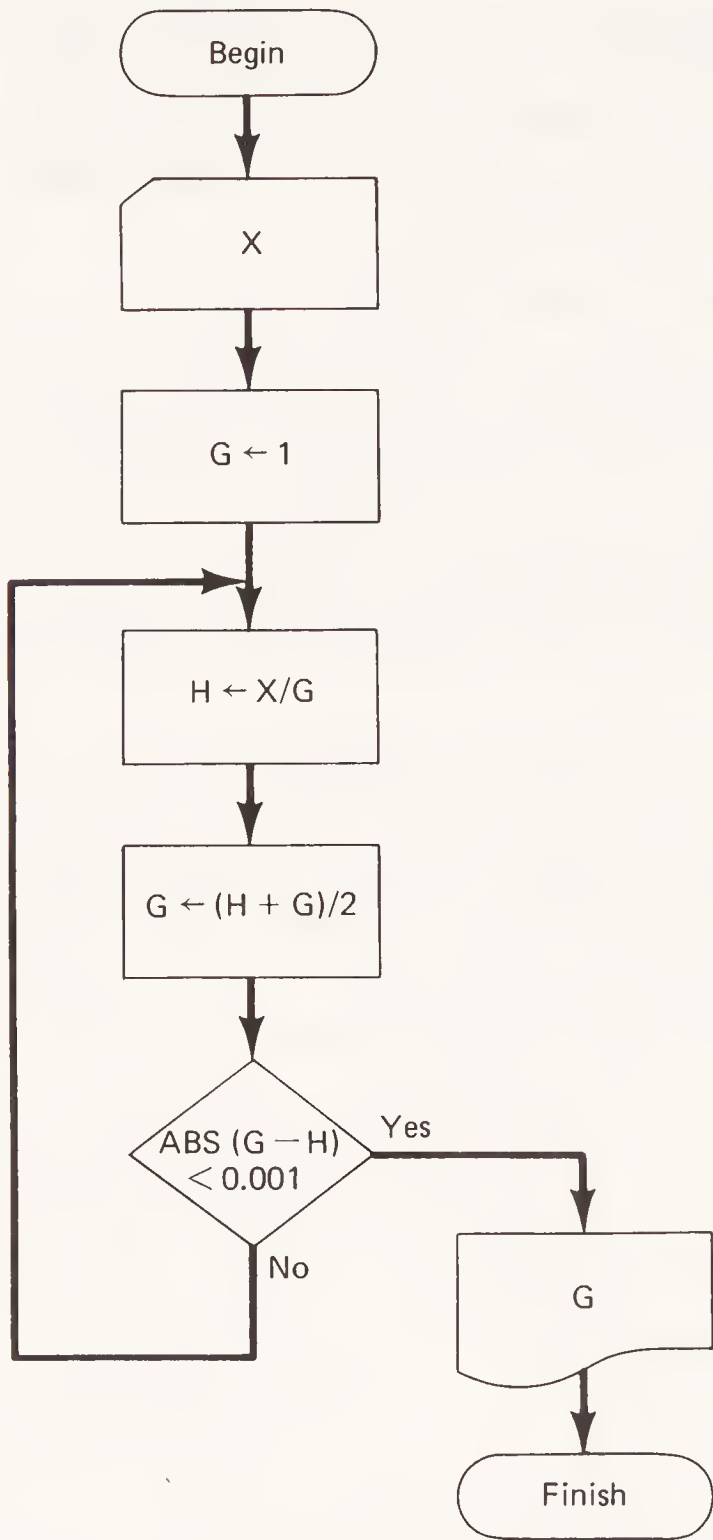


Figure 5-2 Flowchart of a program to compute the square root of X

blocks are “Entry” and “Return.” The variable following the word “Entry” is the one whose square root is to be taken; the value of the square root is that of the variable in the “Return” block.

Let’s consider how the flowchart in Figure 5-3 would utilize the subprogram in Figure 5-4 for computing the square roots. The function SQR is “called” in two places: blocks 5 and 7. The computer’s logic is such that when a function is called, it remembers the place from which the call originated so that it can return there upon completion of the function. Thus, a function can be called as many times as needed. The computer will transfer control to the function, perform the indicated computations, and upon completion return to the original place in the program from which the call originated.

An integral part of this transfer to and from a subprogram is the communication of values. For the routine in Figure 5-4, the value whose square root is to be computed must be communicated to the function. Upon completion, the value

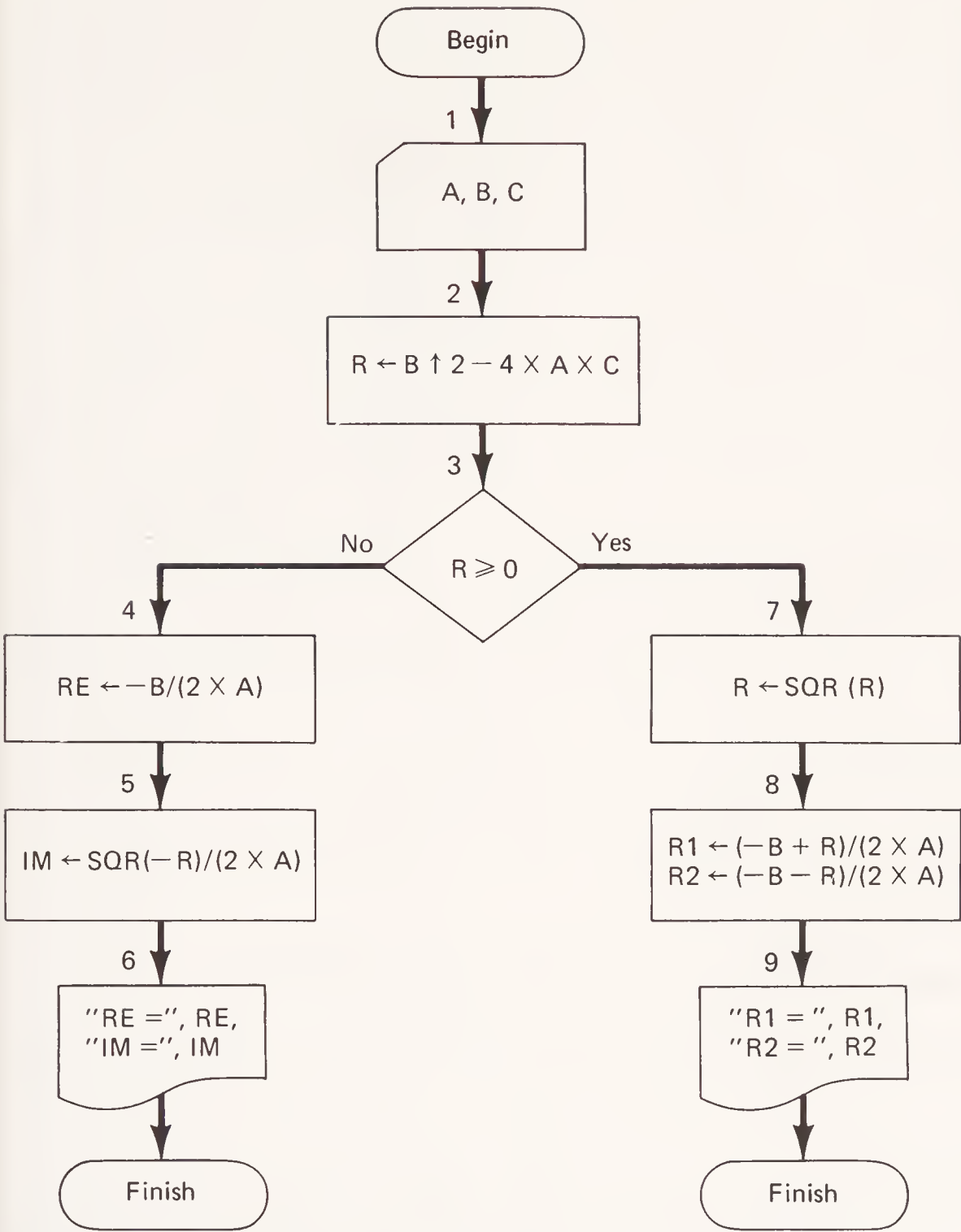


Figure 5-3 Algorithm to compute roots of a quadratic equation

of the square root is to be “returned” to the calling program. The mechanics are such that the value of the argument of the function in the calling program is used as the value of the variable following the word “Entry” in the subprogram.

For example, the function reference in block 7 of Figure 5-3 is

$SQR(R)$

The current value of R is used as the value of X in executing the function in Figure 5-4. The value of G in the function is then used in place of SQR (R) in evaluating the expression in block 7 of Figure 5-3.

In block 5, the function reference

$SQR(-R)$

causes the expression

$-R$

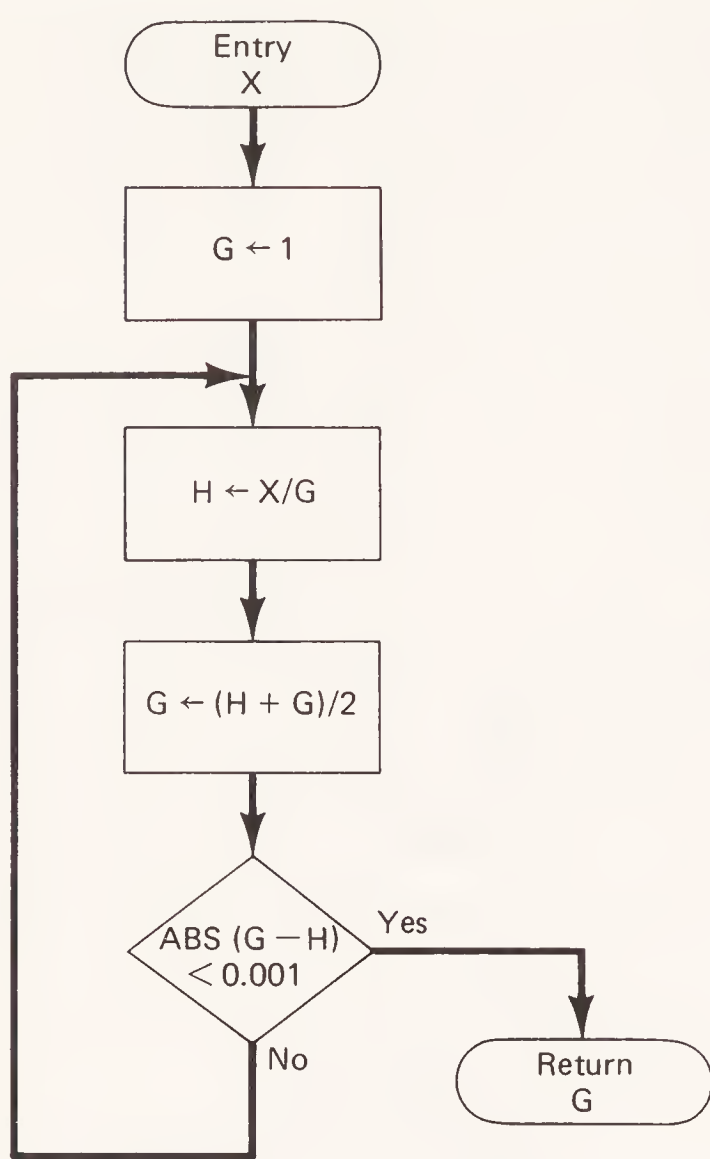


Figure 5-4 Reference flowchart for the square-root computation

to be computed and its value used for X in the flowchart in Figure 5-4. The computed value of G is again used in place of $SQR(-R)$ in evaluating the expression in block 5.

Most programming languages permit any number of values to be transferred to a function. For example, suppose we want to find the maximum value of three variables, say, P , Q , and R , and multiply the result by 1.4. Postulating a function MAX for this purpose, a possible calling statement could be

$$RESULT \leftarrow 1.4 \times MAX(P,Q,R)$$

In this case, we have three arguments. The corresponding reference flowchart for MAX is shown in Figure 5-5. The three variables in the “Entry” block, A , B , C , are given the values of P , Q , and R at the time function MAX is called. The computed value of G is then used in evaluating the expression in which MAX appears.

Arrays may also be used as arguments. Suppose the largest value in array X containing N elements is to be computed and multiplied by 1.4. If the entire array is to be available to the function, the name of the array is generally written in the calling statement *without* subscripts, for example,

$$RESULT \leftarrow 1.4 \times ARRAYMAX(X,N)$$

The reference flowchart for $ARRAYMAX$ is illustrated in Figure 5-6. The values

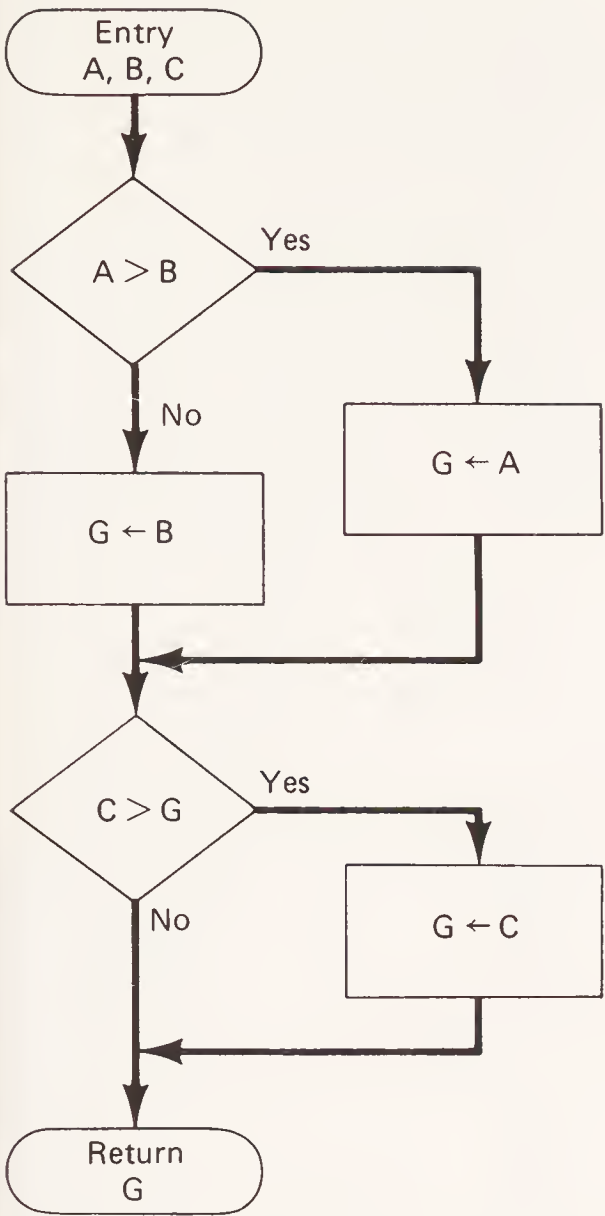


Figure 5-5 Reference flowchart for function MAX

of array X in the calling flowchart are used for array A in the flowchart for computing G, which in turn is used in computing the expression in which the function call appears.

Function ARRAYMAX as written is for locating the largest element from an array. The first argument is the name of the array written without a subscript, which designates that the entire array is available to the function. If an array name written with a subscript appears in an argument list, this designates that only the value of that respective element is available to the function. For example, if array V contains only three elements, function MAX in Figure 5-5 could be used as follows:

$$\text{RESULT} \leftarrow 1.4 \times \text{MAX} (V_1, V_2, V_3)$$

Alternatively, variables and expressions can be used as the subscripts, as illustrated by the following example:

$$\begin{aligned} J &\leftarrow 1 \\ \text{RESULT} &\leftarrow 1.4 \times \text{MAX} (V_J, V_{J+1}, V_{J+2}) \end{aligned}$$

Functions may have more than one RETURN statement. For example, the function ARRAYMAX is probably not being used properly if it is called with

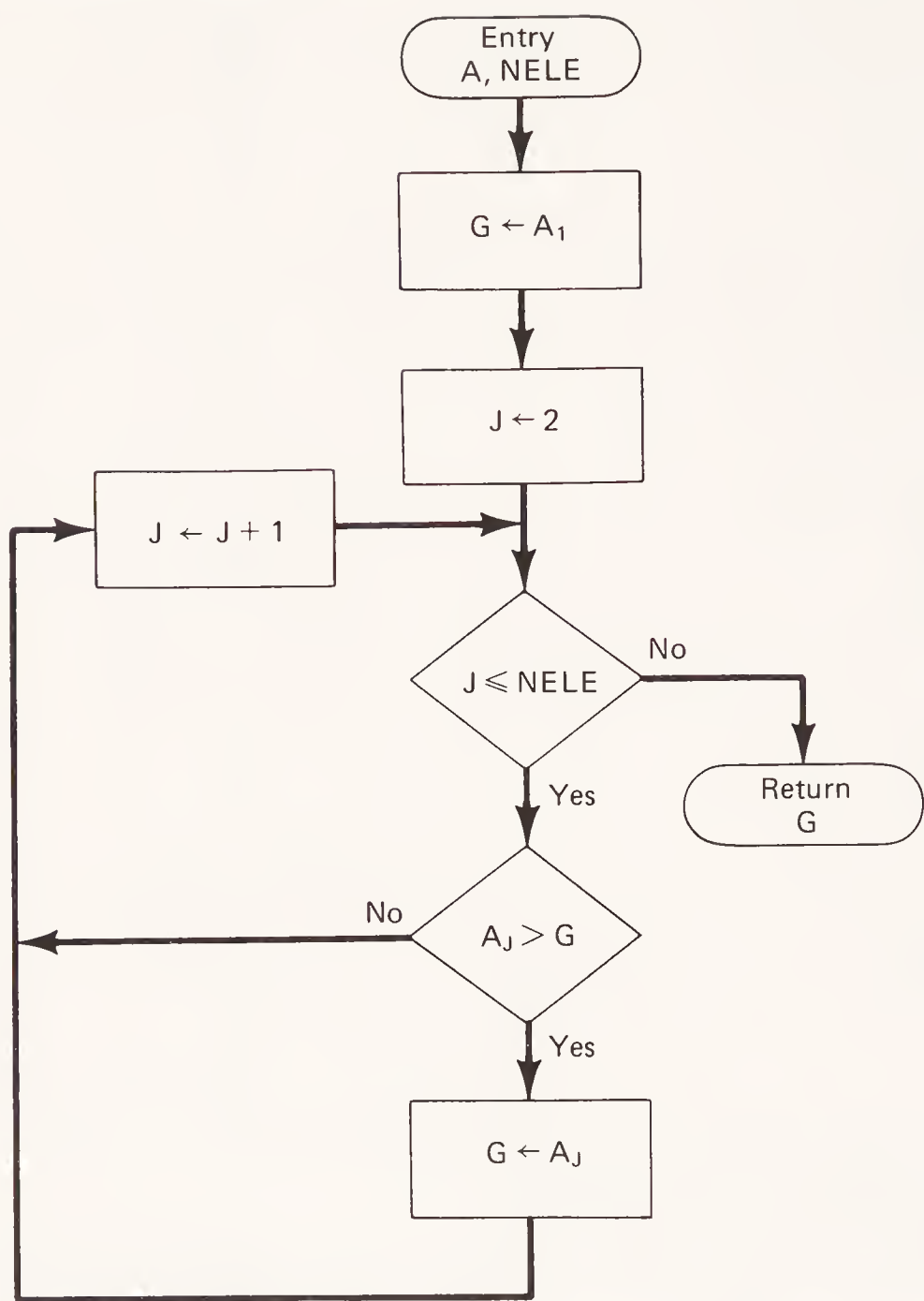


Figure 5-6 Reference flowchart for function ARRAYMAX

the number of elements N assigned the value less than or equal to zero. In this case, suppose we write our reference flowchart such that the message

FUNCTION ARRAYMAX CALLED FOR
AN ARRAY WITH xxx ELEMENTS

is printed, where xxx is the number of elements. In order to allow processing to continue, we shall assign the value zero to the maximum value and continue processing. The resulting reference flowchart is shown in Figure 5-7.

The flowchart in Figure 5-7 also illustrates that a function may contain input/output statements. Furthermore, a function may contain a terminal block. That is, if we had chosen to terminate execution upon encounter of a zero or negative number of elements, block 9 in Figure 5-7 could be replaced by a “Finish” block. In general, however, a function must contain at least one RETURN block. This is because a function is always called by the appearance of the function name in an expression. If the function called does not contain at least one RETURN block, evaluation of the expression cannot be completed.

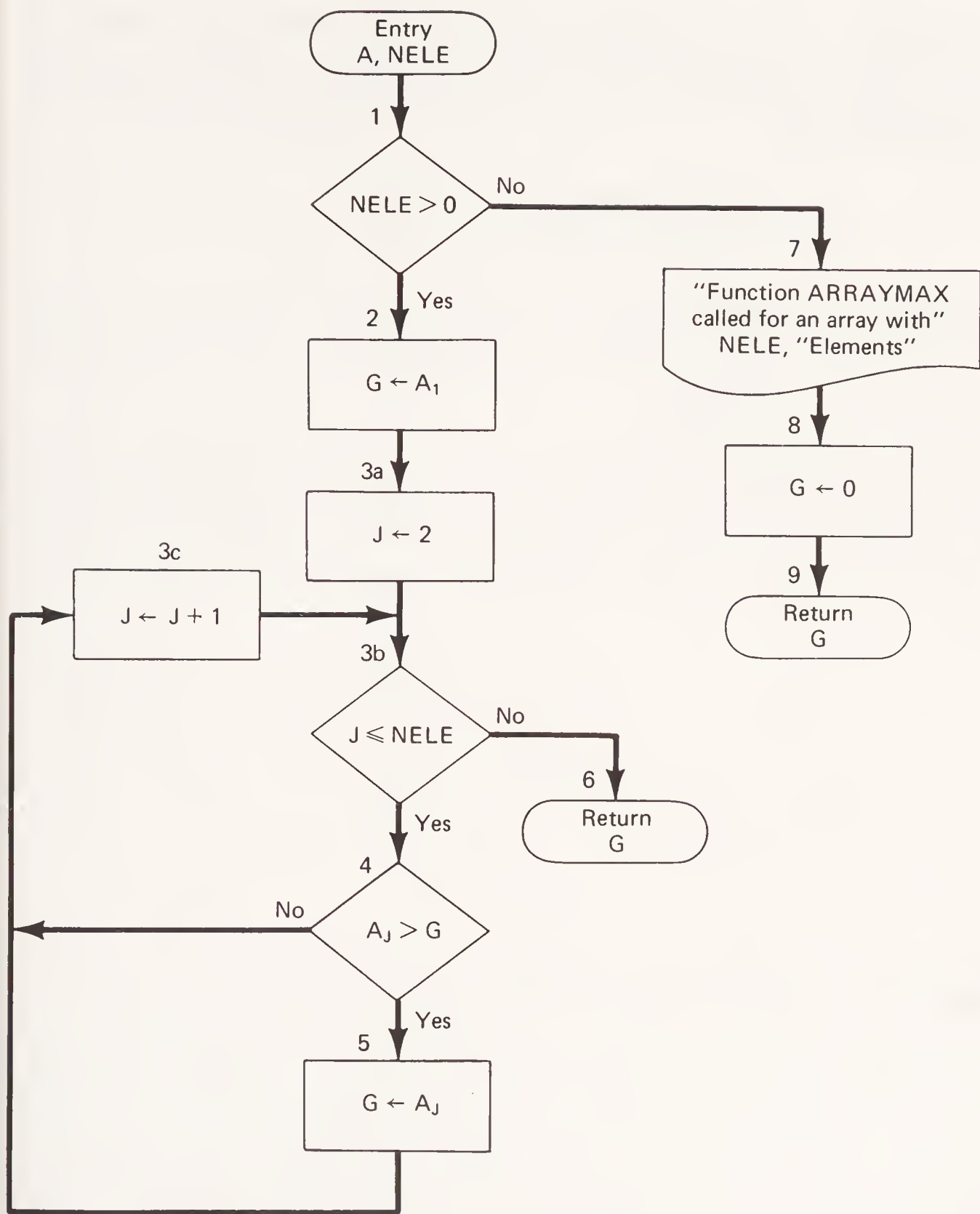


Figure 5-7 Function ARRAYMAX with check for an array with zero or a negative number of elements

5.2. PROCEDURES OR SUBROUTINES

As presented in the previous section, functions compute a single value from one or more arguments, which may be either simple variables or arrays. In many cases, we want to compute more than one answer, or we might want to accomplish some objective such as arranging the elements of an array in ascending or descending order.

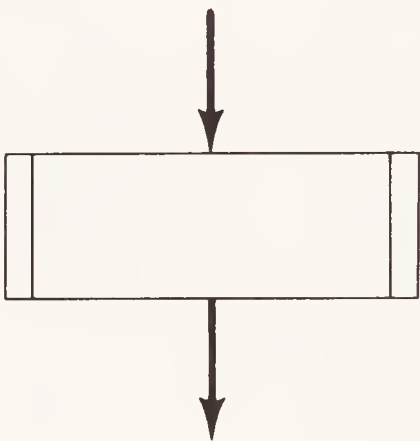
Particularly in the latter case (that is, ordering an array), the use of a function in an expression is not appropriate. If no value is returned, evaluation of an expression is not feasible.

In these cases, a procedure or a subroutine is more appropriate. These are called by special statements in the programming language, not by their use in an expression. The calling statements vary from one language to the next, but we shall adopt the following:

```
PERFORM ORDER (ARRAY, NELE)
CALL ORDER (ARRAY, NELE)
```

where ORDER is a procedure that is used to rearrange the elements of array ARRAY containing NELE elements in ascending order. Upon call, ARRAY contains the elements in non-ascending order. Upon return, ARRAY contains the elements in ascending order.

To illustrate the mechanism used, Figure 5-8a contains a program that reads two arrays, calls ORDER to arrange them in ascending order, and prints the results. The reference flowchart for ORDER is shown in Figure 5-8b. Note in Figure 5-8a that we use a special block to indicate reference to a subroutine or procedure.



In allocating memory for the program in Figure 5-8a, storage locations are reserved for every variable appearing in the program. Assuming for convenience that only six elements are reserved for each array and that the first location available for variable storage is location 1022, the assignments would be as illustrated in Table 5-1. Assuming that location 1702 is the first storage location available for variable storage for ORDER, the assignment of storage for variables is as shown in Table 5-1.

In languages such as Fortran that implement procedures via separate subprograms, no storage locations are reserved for array X nor variable N in subroutine or procedure ORDER. The reason is that the communication between the main program and the procedure is via addresses instead of values. That is, upon execution of the call in block 3 of Figure 5-8a,

```
CALL ORDER (A, NA)
```

the addresses 1022 (first element in array A) and 1034 (variable NA) are transferred to ORDER to be used for array X and variable N in that subprogram. Upon executing block 4, addresses 1028 and 1032 are transferred to ORDER. Thus, ORDER is effectively operating on the variables that appear in the calling statement. The respective variables in ORDER are dummies; that is, they are not assigned storage locations.

Table 5-1. Assignments of Storage for Variables

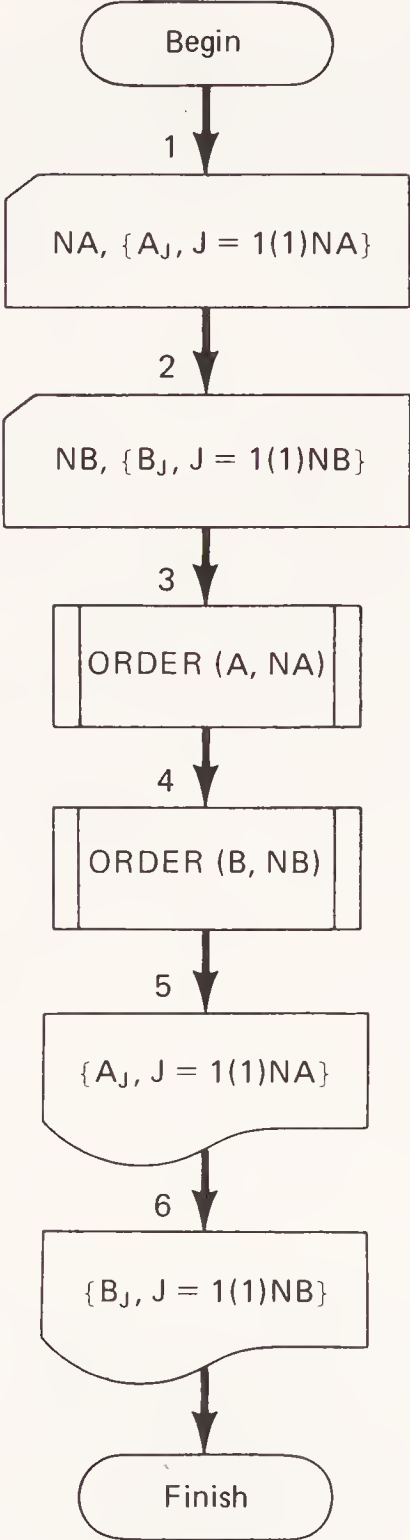
STORAGE OF VARIABLES FOR MAIN PROGRAM		STORAGE OF VARIABLES FOR ORDER	
Location	Variable	Location	Variable
1022	A ₁	1702	N ₁
1023	A ₂	1703	1
1024	A ₃	1704	J
1025	A ₄	1705	M
1026	A ₅	1706	K
1027	A ₆	1707	T
1028	B ₁		
1029	B ₂		
1030	B ₃		
1031	B ₄		
1032	B ₅		
1033	B ₆		
1034	NA		
1035	J		
1036	1		
1037	NB		

A second point to note is that communication between the calling program and procedure ORDER in Figure 5-8 is *only* via the arguments. Variable J appears in both the main program and in ORDER. However, location 1035 is used for J in the main program and location 1704 is used for J in ORDER. Therefore, changes in variable J in ORDER are *not* reflected in the storage locations for the main program. Similarly, duplicate storage locations are reserved for the constant 1, namely, location 1036 in the main program and location 1703 in ORDER.

The point is that, in our current example, only variables appearing in the argument list are available to the subprogram. This statement applies to functions as well as subroutines or procedures. We shall take a closer look at the problem of communication with functions and procedures in a later section, and will discuss other ways of communicating.

Although only the values of the array in Figure 5-8 were changed, the same principles apply to simple variables. Suppose a procedure is to be written that takes three simple variables, say A, B, and C, and places the largest value in A, the second largest in B, and the smallest in C. A typical block referencing our procedure, which we shall call JUGGLE, could be



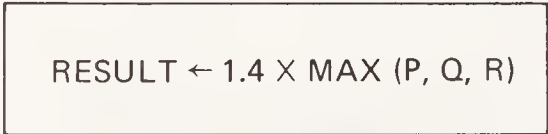


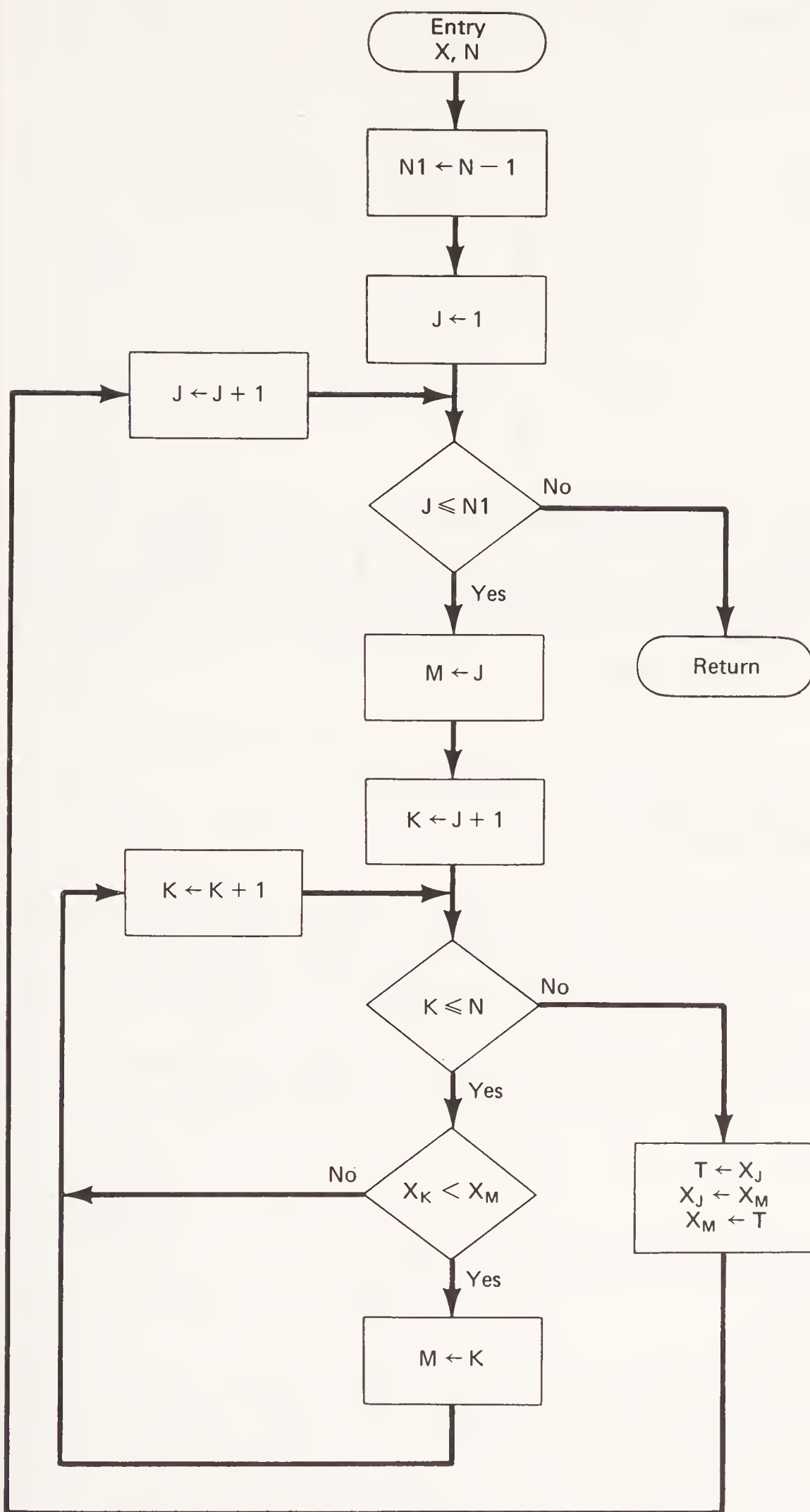
a. Calling program

Figure 5-8 Use of procedure or subroutine

The reference flowchart for procedure JUGGLE is given in Figure 5-9. Again, the storage location for variable A in the main program is used for variable X in JUGGLE; the location for variable B is used for variable Y; and the location for variable C is used for variable Z.

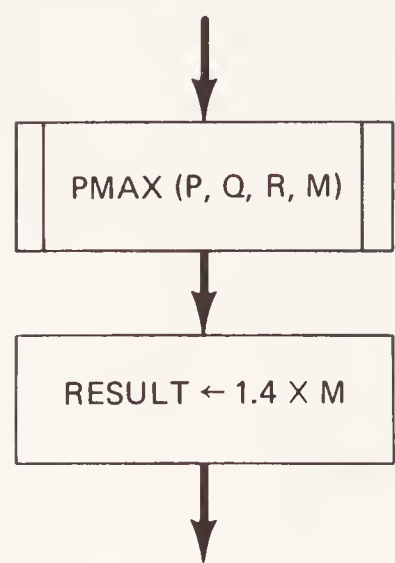
In many respects, a procedure can be used to perform any task for which a function could be used. For example, the function MAX in Figure 5-5 could be replaced by the procedure PMAX. As illustrated in the reference flowchart in Figure 5-10, the maximum of the three values now appears in the argument list. However, to compute 1.4 times the maximum value, the single block





b. Subroutine or procedure ORDER
Figure 5-8 (Continued)

188 using function MAX must be replaced by the two blocks



While the procedure could be used, the function is much more convenient when only a single value is to be computed. A more pragmatic advantage of the function is that virtually all programming languages permit the use of expressions as arguments, which is not permitted for procedures in many implemented versions of programming languages.

5.3. COMMUNICATION WITH FUNCTIONS AND PROCEDURES

In our previous discussions, we have touched upon the use of arguments to communicate with procedures and the process of calling by address. In this section, we shall make a more thorough examination of communication with functions and procedures.

In general, variables appearing in a program may be any one of three types:

- 1. Local
- 2. Arguments
- 3. Global

In the case of arguments, we have two types of use:

- 1. Call by address
- 2. Call by value

In the ensuing paragraphs, we shall examine the role of each.

Local

These variables are used exclusively within the function or procedure. Separate storage locations are established for these variables, and they are accessed only from within the function or procedure itself. Names of these variables may generally be the same as that of variables in other segments of the program.

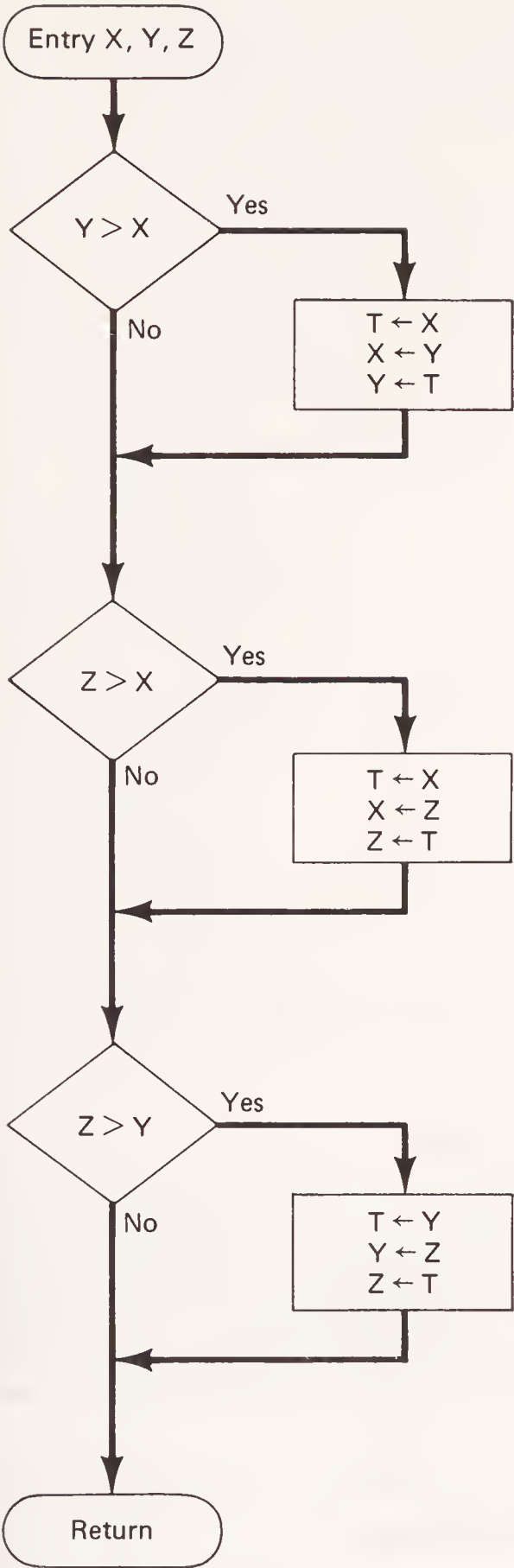


Figure 5-9 Reference flowchart for procedure JUGGLE

Since separate storage locations are assigned, however, changing the value of a local variable in a function or procedure in no way affects the value of the variables in other parts of the program. In effect, a local variable is isolated to within the function or procedure.

Global

In the simplest implementations, these variables are accessible throughout the program, including functions and procedures. That is, if a variable is declared

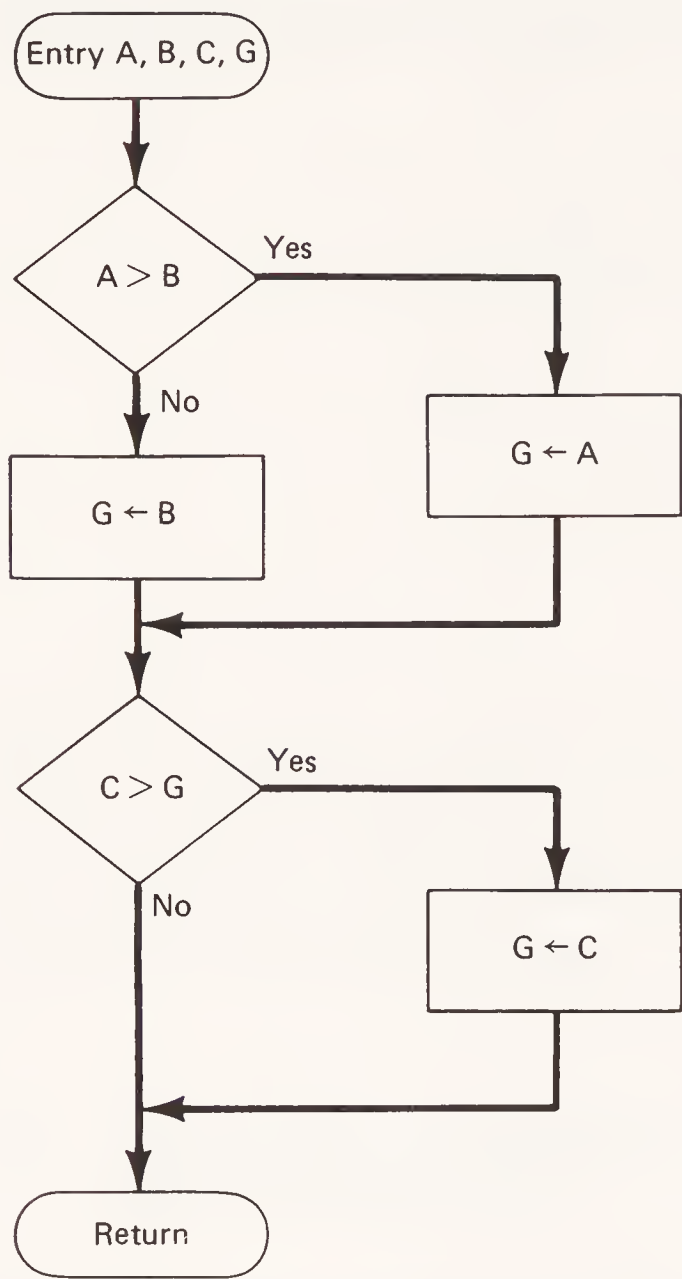


Figure 5-10 Reference flowchart for procedure PMAX

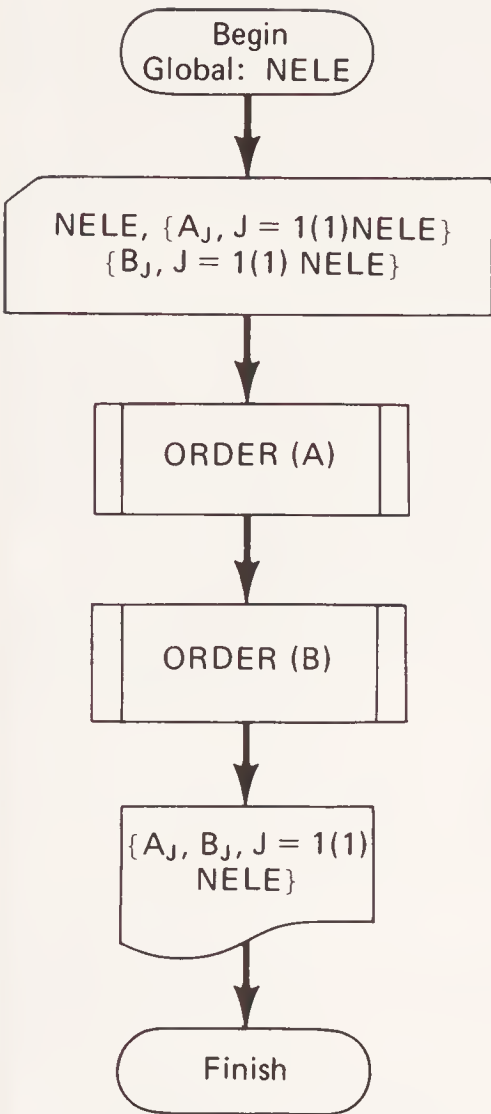
global, use of that variable in one part of the program refers to the same variable in the remainder of the program. This could effectively be accomplished by using the same argument name in the function or procedure as is used in the calling program. However, use of the global declaration has two advantages:

- 1. It reduces the length of argument lists, which can get out of hand.
- 2. It reduces the computational duty in transferring to the subprogram, since the address or value does not need to be transferred.

In general, the global declaration is preferred over the use of arguments whenever a function or procedure is always called with the same variable as the argument.

For example, consider the array-ordering program in Figure 5-8. Furthermore, suppose the number of elements in array A is always the same as in array B. Then a simple variable, say, NELE, could be used to store the number of elements. By declaring NELE global, it could be removed from the argument lists, as illustrated in Figure 5-11. Since, however, ORDER is called once using array A and once using array B, use of an argument for these is much more convenient than a global declaration.

By declaring a variable global, we essentially mean that we know what storage



a. Calling program (NELE declared global)

Figure 5-11 Illustration of the use of global variables

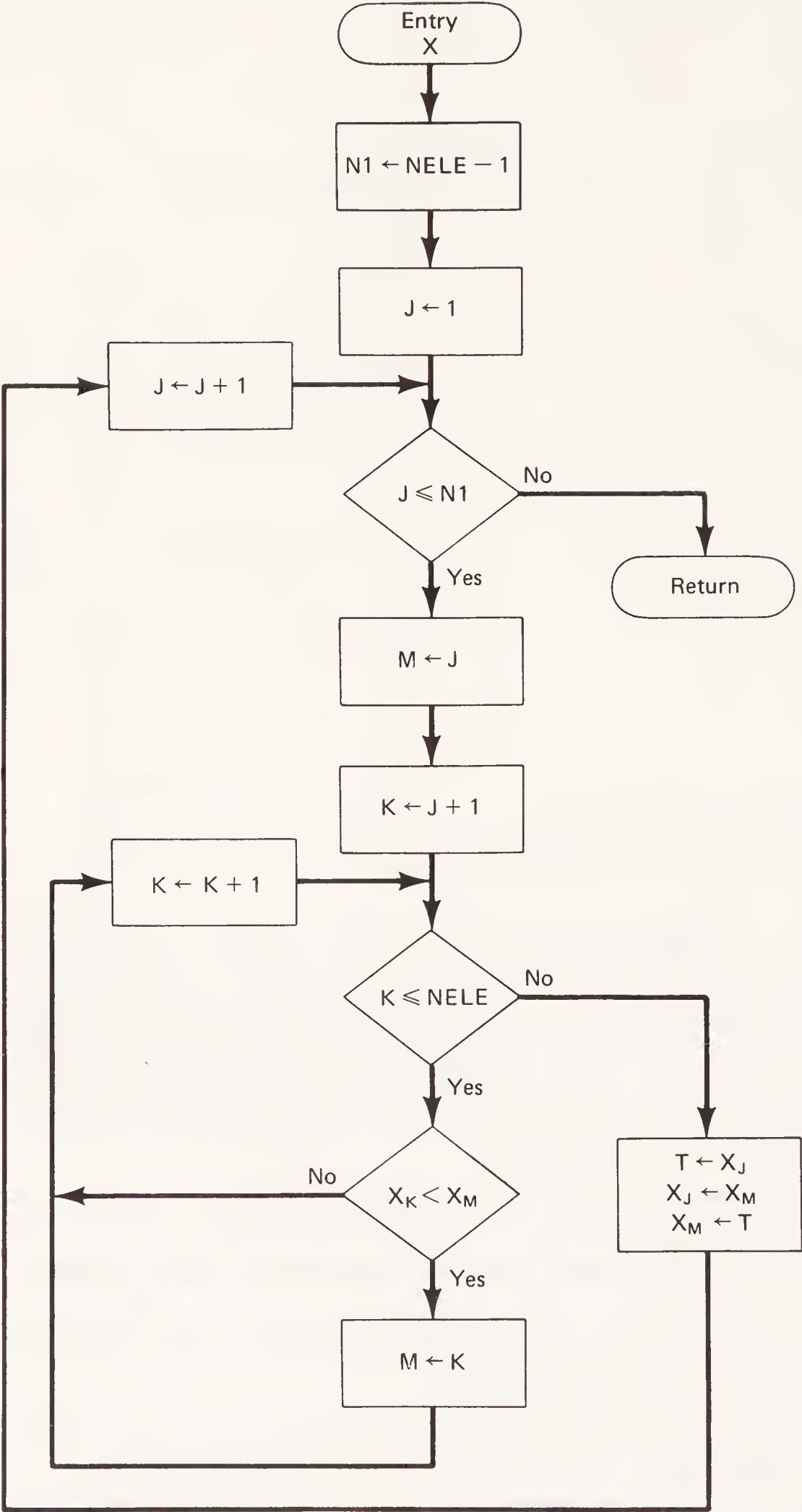
location is used for this variable throughout the program. When variables appear in the ENTRY block, it is implied that the storage locations are not known at compile time and that the addresses will be supplied when the function or procedure is called. These are mutually exclusive conditions, which exclude the appearance of a global variable in the argument list of a function or procedure within which this variable is declared global.

In Fortran, the COMMON declaration is similar to a global declaration but different in that variables may be declared “global” or “in common” between the main program and selected subprograms or only between selected subprograms. That is, some variables may be “in common” between the main program and subprogram A, others between the main program and subprogram B, and still others only between subprograms A and B.

COMMON also permits the programmer to change the name of a location “in common” between subprograms. In effect, a “common” storage location is established but is referred to by one name in the main program, by another name in one subprogram, and perhaps by yet another name in another subprogram.

Arguments

Every example presented prior to this section used arguments as the communication mechanism between the main program and the function or procedure. We



b. Procedure ORDER with NELE declared global
Figure 5-11 (Continued)

have also discussed the mechanism involved in calling by address. In calling by value, the actual value of the variable is transferred to the subprogram instead of the address of the variable in the calling program. Calling by value is essentially a

one-direction communication: a value is transferred to the subprogram, but no transfer back to the main program occurs for this particular argument.

One place where calling by value would certainly be appropriate is when an expression is used as an argument, for example,

$$R \leftarrow \text{SQR} (B \uparrow 2 - 4 \times A \times C)$$

Since the argument of the function SQR is an expression, its value is made available to the function and upon return there is no need to make available a resulting value of the argument to the main program. The only value returned is that to be used in evaluating the expression in which the SQR function appears.

If inappropriately used, the process of calling by address can yield some peculiar and often distressing results for the unwary user. For example, suppose the calling block for JUGGLE (Figure 5-9) is



where F and Q have the values 3.4 and 7.9 at the time JUGGLE is called. Upon return, the storage is arranged as follows:

Variable or constant assigned to storage location	Value upon call	Value upon return
F	3.4	7.9
2	2.0	3.4
Q	7.9	2.0

The problem is that the storage location assigned for the constant 2.0 now contains the value 3.4. If any statement containing the constant 2.0 is subsequently executed, the value 3.4 is used instead. The resulting confusion should be obvious, and fortunately most programming languages do not use call by address for constants.

Calling by address is especially attractive when arrays are used as arguments. If an array contained fifty elements, calling by value would require (1) communication of each of these fifty values to the function or procedure and (2) the establishment of fifty separate storage locations in the function or procedure to store these values. Calling by address requires only the one storage location in which to store the address. Thus, calling by address is preferred for arrays.

For simple variables, calling by value or calling by address is equally as convenient. One storage location is required, which could be used to store either an address or the value itself. Similarly, one item of information, either the address or the value, is to be communicated.

While calling by address is invariably used for arrays, some programming languages in certain situations use call by value for simple variables that appear in the argument list. To the programmer, the result may not be the same whenever the value of the corresponding argument is changed within the func-

194 tion or procedure. When the call is made by address, this change is reflected in the corresponding variable in the calling program; when the call is made by value, this change is not reflected in the calling program.

5.4. MULTIPLE ENTRY

In the examples presented so far, all reference flowcharts contained only one ENTRY block. Many current programming languages, however, permit several entry points, a feature known as *multiple entry*.

To illustrate the use of the multiple-entry feature, suppose the function relating the value of a resistor element to temperature is given by

$$R \leftarrow 987 + 0.426\ T$$

where T is in degrees centigrade ($^{\circ}\text{C}$). Furthermore, suppose we would like to construct a function which could be called with T in either degrees centigrade or degrees Fahrenheit, the relationship being

$$^{\circ}\text{C} = (^{\circ}\text{F} - 32) / 1.8$$

Using the multiple-entry feature, the reference flowchart of this function is shown in Figure 5-12.

When the temperature in the calling program is available in degrees centigrade, the call would appear as follows:

$$\text{VOLTS} \leftarrow \text{AMPS} \times \text{RC} (\text{TCENT})$$

On the other hand, if the temperature is available in degrees Fahrenheit, the call would be

$$\text{VOLTS} \leftarrow \text{AMPS} \times \text{RF} (\text{TFAR})$$

This function also illustrates a case where different results are obtained

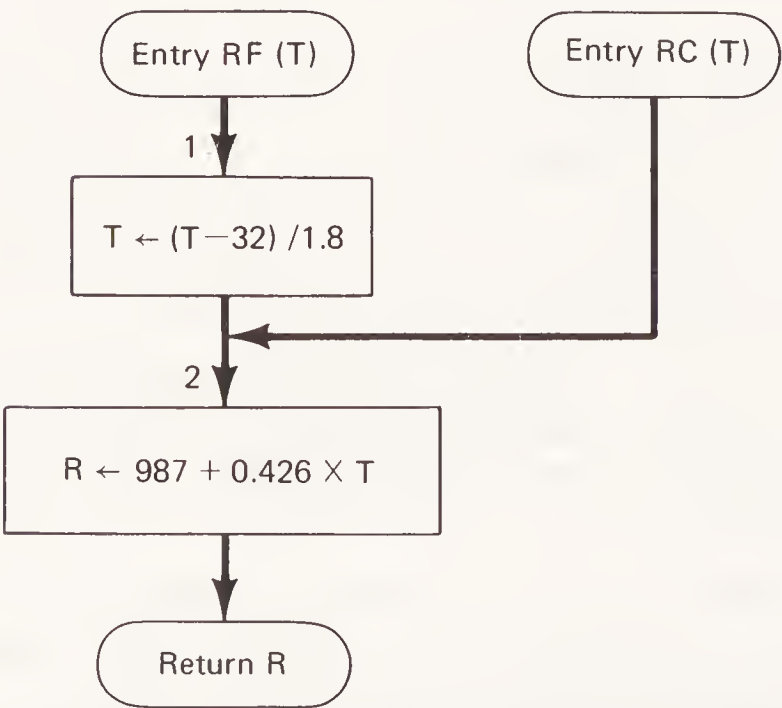


Figure 5-12 Use of the multiple-entry feature

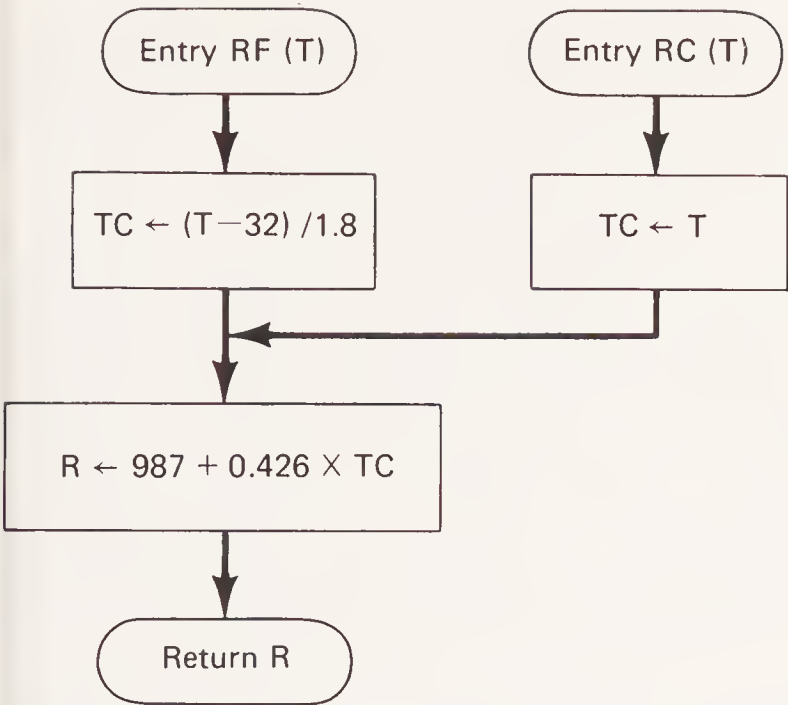


Figure 5-13 Function that will give consistent results for either call by value or by address

depending upon whether the call is by address or by value. If the calling statement is

$$\text{VOLTS} \leftarrow \text{AMPS} \times \text{RF}(\text{TFAR})$$

and the call is by address, TFAR is effectively converted from Fahrenheit to centigrade in block 1 of the reference flowchart in Figure 5-12. But if the call is by value, the contents of the storage location used for TFAR are not changed and TFAR remains in Fahrenheit. Of course, if TFAR is not used again following the call to the function, this is immaterial. But if it is used, then one must know how the call was made.

If one is not sure how the call is made, constructing the flowchart as in Figure 5-13 removes all uncertainty. In this case, TFAR would not be changed if the call were by address or by value.

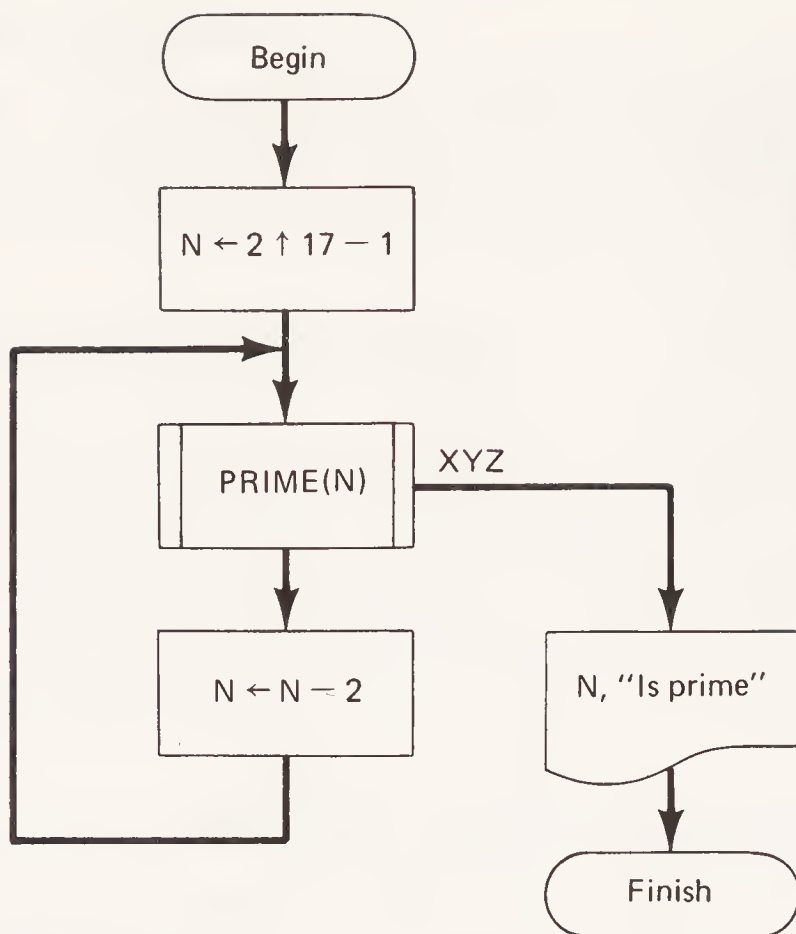
The multiple-entry feature can be used for both functions and procedures.

5.5. MULTIPLE RETURN

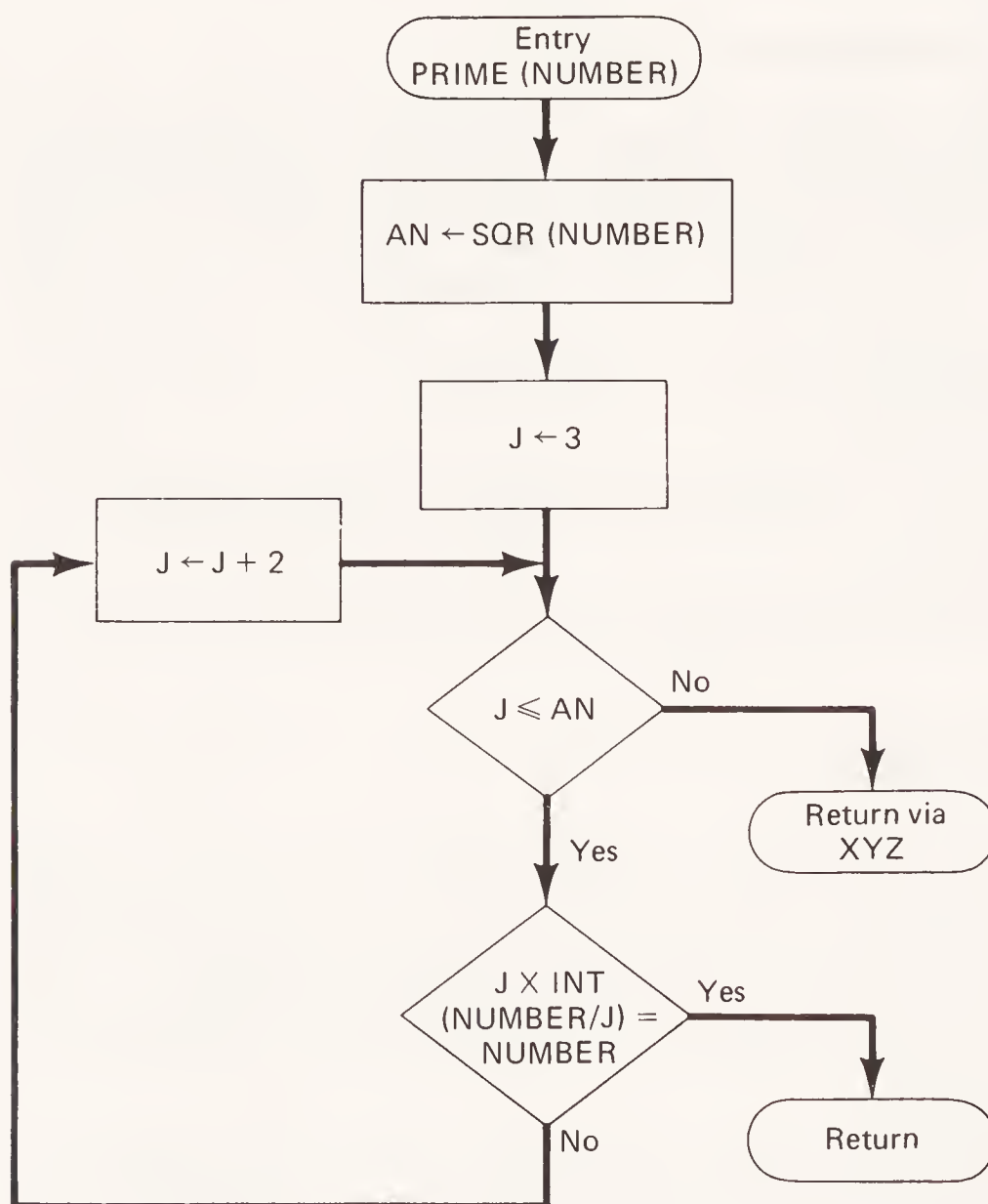
Although we have pointed out that more than one RETURN block is allowed, each RETURN block transferred control back to the same point in the calling program. In some cases, it would be convenient if the RETURN blocks could transfer control back to alternative points in the calling program. This feature is referred to as *multiple return*.

To illustrate the use of the multiple-return feature, consider preparing a program to determine the largest prime number whose value is less than 2^{17} . In preparing this program, we shall make sure of a procedure that determines if a given number is prime.

Our attack to this problem is to first compute 2^{17} and then subtract 1, giving us an odd integer. If it is not prime, we then subtract 2, giving us another odd



a. Main program



b. Procedure

Figure 5-14 Use of the multiple-return feature

integer. This is continued until a prime number is found, as illustrated by the flowchart in Figure 5-14. Since no even integers are encountered, procedure PRIME does not need to try dividing by 2 and can start by trying 3.

In our flowchart, we shall indicate use of the multiple-return feature by using a procedure block with more than one exit, such as in Figure 5-14a. All but one of the exits will be labeled. When a normal RETURN block is executed, return is via the unlabeled exit from the procedure block. When a “Return via” block is executed, the exit from the procedure block is via the one so labeled.

Figure 5-14 also illustrates a case where a global declaration can be used instead of placing a variable in the argument list. If NUMBER were used in place of N throughout the main program, then NUMBER could have been declared global and deleted from the argument list. Alternatively, N could have been used in place of NUMBER throughout procedure PRIME and declared global. In either case, procedure PRIME would be left with no arguments.

Multiple returns are acceptable for procedures but not for functions. Since a function is always referenced via its appearance in an expression, return must always be to this expression. Therefore, the multiple-return feature cannot be used for functions.

EXERCISES

5-1. Exercise 4-14 called for the evaluation of a polynomial of the following type:

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

For this exercise, the main program should read the coefficients and store them in an array, read the value of b , call a function to evaluate $f(b)$, and then print b and $f(b)$. In programming the function, use the more efficient approach suggested in Exercise 4-14.

5-2. Suppose we select any positive integer number, say, 116. From this number we generate a new number by summing the squares of the digits, for example,

$$1^2 + 1^2 + 6^2 = 1 + 1 + 36 = 38$$

Suppose we continue this process:

$$\begin{aligned} 3^2 + 8^2 &= 9 + 64 = 73 \\ 7^2 + 3^2 &= 49 + 9 = 58 \\ 5^2 + 8^2 &= 25 + 64 = 89 \\ 8^2 + 9^2 &= 64 + 81 = 145 \end{aligned}$$

If this process is applied to any number and continued, the number 145 will be encountered *unless* the number 1 is encountered in the sequence. Continued application of this procedure generates the repetitive sequence 145, 42, 20, 4, 16, 37, 58, 89, 145.

Suppose we want to write a program that accepts any positive number and determines how many steps are required before the number 145 or the number 1 is encountered. This program is to use a function or procedure that accepts any positive integer number and computes the sum of squares of its digits. Prepare the flowchart for the complete program including the function or procedure.

5-3. If the procedure described in the above exercise is applied to the numbers 10 or 100, the sequence converges to 1. Modify the program to determine what numbers if any between 10 and 100 lead to a sequence in which a 1 appears.

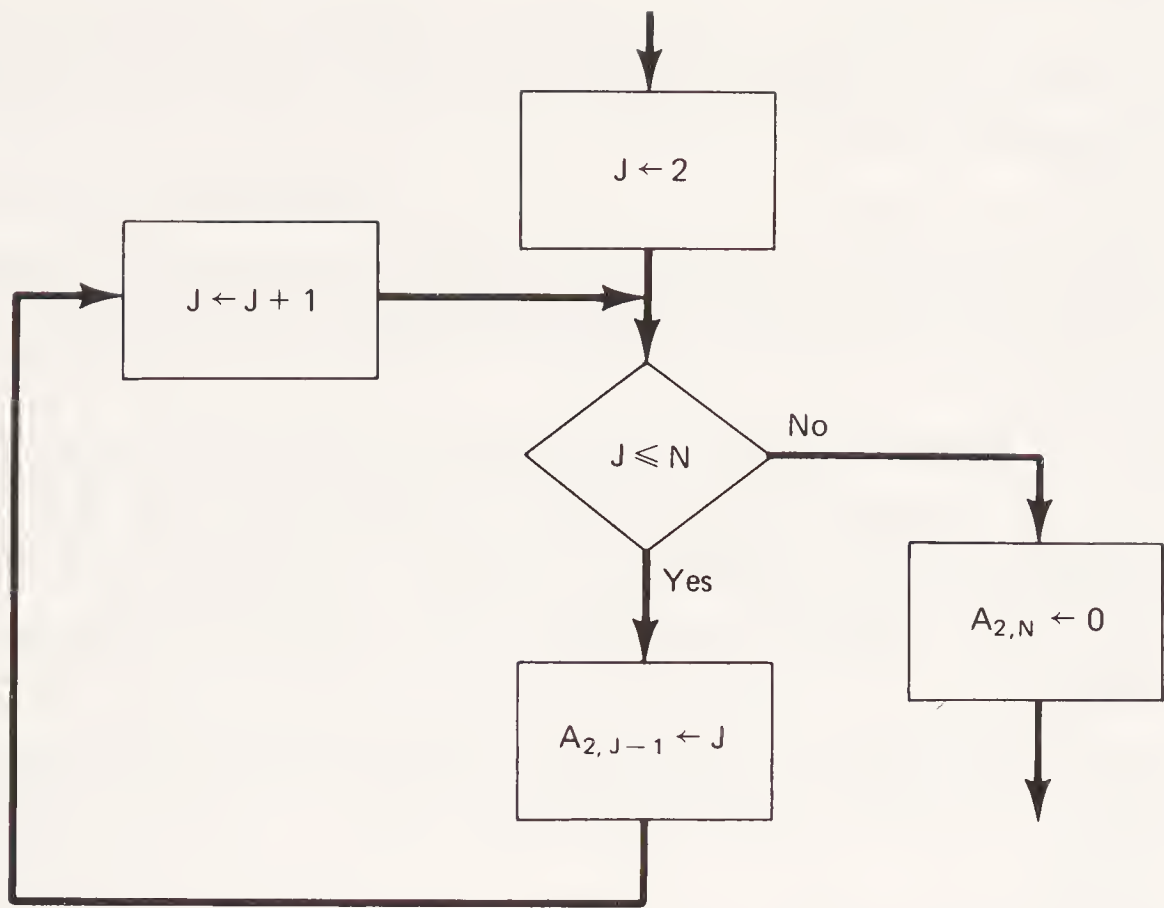
5-4. Suppose we write a function SUBSTR(X,Y,L,M,N) to locate a substring of characters, where X is a single-subscripted array containing L characters (stored one per element) to be scanned for the substring, Y is a single-subscripted array containing the characters in the substring to be located, M is the index of the character in X at which the search is to commence, and N is the number of characters in the substring. The function is to determine the index of the first character in array X that is in the desired substring. If the substring is not found, a zero should be returned.

Prepare the flowchart for this function. Then prepare the flowchart for a program using this subprogram that reads a sentence and counts the number of occurrences of the word “the.”

5-5. Table 4-6 presented an example of text editing using a linear list. Suppose a double-subscripted array A is used for the string element, where $A_{1,J}$ gives the character in the J storage location and $A_{2,J}$ contains the index of the element containing the next character in the string. Furthermore, simple variable B contains the index of the first element in the string; simple variable E contains the index of the last element in the string, and simple variable U contains the index of the first element in the unused string. At the start of the program, U is set to 1, B and E are set to zero, and array A is initialized with the following loop (N is the maximum value allowable for the second subscript of array A). Variables A, B, E, and U should be declared global. Implement this in a procedure.

To facilitate the manipulations involving the linear list, the following routines should be implemented either as functions or procedures:

1. Add a character to the end of the character string stored in the linear list.
2. Delete the K characters beginning with the character stored in element L of array A from the string. The necessary changes must be made so that the storage locations for the characters being removed are added to the unused list.



- 3. Locate a desired substring within the string stored in the linear list. This routine should correspond to the function SUBSTR described in Exercise 5-4.
- 4. Copy a substring of M characters from the linear list (starting with element L) into another array.
- 5. Add a substring containing N characters to the string stored in the linear list beginning with element L of array A . The required elements must be taken from the unused list.

Using *only* these routines to reference variables A , B , E , and U , prepare the flowchart for a program to perform the manipulations in Tables 4-6 and 4-7 upon the sentence stored in the linear list. The program should read the original sentence, perform the manipulations, and print the final sentence.

5-6. Prepare the necessary flowcharts to implement the following routines as functions or procedures:

- 1. Convert a fractional number (that is, less than 1.0) into a binary number, storing the digits in an array (refer to Exercise 3-33).
- 2. Convert a sequence of binary digits comprising a binary fraction into a decimal fraction. The digits are available in an array, with the binary point assumed immediately to precede the first digit. The method for making this conversion is given in Chapter 6.

Using these routines, write a program that reads a decimal fraction between 0.5 and 1.0, converts it to a binary fraction containing N digits, converts the binary fraction back to a decimal fraction, and prints the result. Run the program for the decimal fraction 0.6 with values of 5, 10, 15, and 20 for N .

5-7. Modify the flowcharts in the previous exercise so that they apply to conversions between decimal and any other number system when the base is specified at execution time.

5-8. Prepare the flowchart for a function or procedure that accepts an integer number and converts it to character data, storing one character in each array element. Use this function or procedure in preparing the flowcharts for Exercises 4-32, 4-33, and 4-34.

5-9. Figure 4-3 presented the basic version of a stock-inventory program. In this exercise, we want to make several extensions to this program. The first step is to prepare flowcharts for the functions or procedures to accomplish the following:

1. Arrange array STOCK in ascending order, rearranging array QUAN accordingly.
2. Locate the index of a specific item number in the ordered array stock using the approach in Figure 4-7 and update the corresponding entry in QUAN.

Using these routines, prepare the flowchart for a program that reads an initial inventory in random order, updates this inventory for items received and for items shipped, and then prints the final inventory in ascending order of the stock numbers.

5-10. The greatest common divisor of two numbers is the largest integer number that will divide (with no remainder) each of the two numbers. Although there are more efficient methods, the simplest approach is to assume that 1 is greatest common divisor, and then try 2, 3, 4, etc., until the smaller of the two numbers is reached. To determine the greatest common divisor of three numbers, find the greatest common divisor, say, N , of two of the numbers, and then find the greatest common divisor of N and the third number. Prepare the flowchart for a function to compute the greatest common divisor of two numbers, and then prepare the flowchart for a main program that uses this function to compute the greatest common divisor of three numbers.

5-11. The number 9 has some interesting properties. For example, suppose we begin with a number that is a multiple of 9, for example, 1,953 ($= 217 \times 9$). If we sum the digits of this number, the result will also be a multiple of 9. That is, $1 + 9 + 5 + 3 = 18$ ($= 2 \times 9$). If we then sum the digits of the resulting number, we obtain another number that is a multiple of 9. That is, $1 + 8 = 9$. In fact, if the process is continued long enough the number 9 is obtained.

Prepare the flowchart for a function that accepts an integer number and sums its digits. Then prepare the flowchart for a program using this function that applies this procedure recursively until the number 9 is obtained, printing the numbers encountered in the process. Note that if the sum of digits is ever less than 9, the original number was not a multiple of 9.

5-12. Suppose we generate a sequence of numbers according to the following rule:

1. Start with any number of your choosing.
2. Halve this number if it is even; triple it and add one if it is odd.
3. Repeat the above step on the resulting number, a process that can be continued indefinitely.

To illustrate, suppose we start with the number 36. The generated sequence is 36, 18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ... Notice that eventually the series begins to repeat the 4, 2, 1 sequence indefinitely. Does this hold for any starting number? No one has proved that it does, nor has a counter-example been found.

Prepare the flowchart for a function that, given a number in the sequence, computes the next number in the sequence. Then prepare the flowchart for a main program that reads the starting number, generates the sequence, and counts the number of entries required until a 1 is encountered in the sequence.

5-13. Prepare the flowchart for a program to implement the following operations:

1. Begin with any number j of three digits in which the difference between the first and last digits exceeds 1.
2. Form a new number k by reversing the order of the digits in j .
3. Compute the difference $d = j - k$.
4. Form the new number m by reversing the digits in d .
5. Add m and d to obtain the desired result, which is printed.

If performed properly, the result will always be 1,089.

In solving this exercise, prepare the flowcharts and use the following routines as functions or procedures:

1. Given any integer number, compute and store the individual digits in a single-subscripted array. That is, given the number 347, the first three elements of the array would be given the values 3, 4, and 7.
2. Given the digits of a number in a single-subscripted array, compute the number and store in a simple variable.

5-14. Figure 2-13 presented the flowchart for a program to determine if a number is a prime. Suppose we implement this routine as a procedure used in a program that, given an integer number, determines if it is one member of a prime twin and, if so, prints the prime twin. A number N is a member of a prime twin if and only if N is prime and either $N + 2$ or $N - 2$ is also prime. For example, the numbers 11 and 13 are prime twins. The program should read a candidate for a member of a prime twin, and if so, the program should print the prime twins. If not, an appropriate message should be printed. For this problem, the multiple-return feature is convenient.

5-15. Suppose we wish to locate the first prime twins whose magnitude exceeds 30,000. First, modify the main segment of the program in the previous exercise so that it is a procedure. Then prepare a program that calls this procedure for 30,001, 30,003, 30,005, etc., until a prime twin pair is found.

5-16. Modify the above problem to determine the first set of prime quadruplets whose magnitude exceeds 30,000. Prime quadruplets is a set of four consecutive primes at intervals of 2, 4, and 2. Examples include 11, 13, 17, 19 and 101, 103, 107, 109.

5-17. Let p be a prime number. Then suppose we form the product of the prime numbers up to and including p and add 1 to the result. That is, if p is 11, the computations are as follows

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 + 1 = 2,311$$

Is the result necessarily a prime number? Clearly 2,311 cannot have as a factor one of the numbers 2, 3, 5, 7, or 11. It could, however, have as factors two primes each of which is greater than p .

Prepare the flowchart for a program to compute the numbers generated in this way starting with the first two primes (2 and 3) continuing through the first eight primes. That is, the first two numbers should be generated as follows:

$$2 \cdot 3 + 1 = 7$$

$$2 \cdot 3 \cdot 5 + 1 = 31$$

For each result, determine if it is prime.

In solving this problem, prepare flowcharts for and use the following routines implemented as functions or procedures:

1. A routine to determine if a given number is prime.
2. A function that, given a prime number, returns the next higher prime number. This routine should call the preceding routine.

6

COMPUTER HARDWARE

Computing machines have two basic characteristics, namely, incredible speed and virtually error-free performance, that enable them to be so useful. Since both of these features stem from the hardware itself, we shall in this chapter examine several aspects relating to computer hardware. We shall first consider the binary number system used by most current computers, followed by how information is represented via this number system. We shall then examine means by which this information can be stored in the computer and finally the instructions that operate upon this information.

6.1. NUMBER SYSTEMS

The Arabic, or decimal, number system has gained virtually universal acceptance in the Western hemisphere and has been cited as a significant factor in the development of our society. This number system basically uses ten digits, 0 through 9, and a special character, the decimal point, coupled with a positional notation to represent numerical values.

For example, the number 1,427 is in reality a notation for the following expression:

$$1 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

That is, the last digit in the number is multiplied by 10^0 (or unity). The exponent of 10 is increased by 1 for each successive position to the left.

Similarly, fractional values are represented by positions to the right of a decimal point. For example, the number 72.419 denotes the following expression:

$$7 \times 10^1 + 2 \times 10^0 + 4 \times 10^{-1} + 1 \times 10^{-2} + 9 \times 10^{-3}$$

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

The exponent of 10 is decreased by 1 for each successive position to the right.

The decimal number system is said to have a base, or *radix*, of 10. We find this system extremely convenient in our day-to-day use primarily because we have been exposed to it since a very early age. The positional notation can, however, be used to represent numerical values in number systems other than the decimal system. The concept is the same; only the base, or radix, is changed.

In the computer field, widespread use is made of three other number systems—the binary, the octal, and the hexadecimal. Most computers basically function on the binary number system, which has only two digits—zero and one. The relationship between values in this system and the decimal system is illustrated in Table 6-1. The base or radix of the binary number system is 10 in the binary system or 2 in the decimal system. In expressing numbers in this text, we shall adopt the convention of subscripting the number whenever the number system being used is not obvious. For example, 101_2 is binary, whereas 101_d is decimal.

To convert from binary to decimal, the simplest approach is to again use the positional expression for the number and evaluate in the decimal system. For example,

$$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_d$$

Fractions are also permitted in the binary system, as illustrated by

$$\begin{aligned} 1011.011_2 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 + 0.125 \\ &= 11.375 \end{aligned}$$

A technique for converting from decimal to binary is given in Appendix E, along with a brief discussion of binary arithmetic. Many programmers, however, use tables such as those in Appendix F to facilitate converting from binary to decimal, and vice versa.

While use of the binary number system as the basis for the computer's internal operations offers several advantages, its direct use as a means for communicating with people has its disadvantages, namely, more 1s and 0s than can be remembered even for very short periods of time. Although conversion to decimal is straightforward, manual conversions of binary numbers with 16 or 32 digits require substantial effort. Basically, we need a number system very similar to the decimal system but for which conversion to and from binary is almost a trivial operation. The octal (base 8) and hexadecimal (base 16) number systems serve this purpose.

Table 6-2 presents the octal and hexadecimal equivalents to the first sixteen decimal numbers. In the octal system, only the eight digits 0 through 7 are used. The hexadecimal system requires sixteen digits, the digits 0 through 9 providing ten of them. Although other suggestions have been made, the customary practice is to use the letters A through F as the remaining digits. This practice is illustrated in Table 6-2.

In our previous discussion of the binary number system, the number 101110101_2 would be converted to the decimal system as follows:

$$\begin{aligned} 101110101_2 &= 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 \\ &\quad + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 373_d \end{aligned}$$

To convert to octal, we need only to examine the binary digits in groups of three from the right, converting each group to octal with the aid of Table 6-2. For 101110101_2 the procedure is as follows:

$$\begin{array}{cccc} 101 & 110 & 101 & \text{(binary)} \\ 5 & 6 & 5 & \text{(octal)} \end{array}$$

The resulting octal number can be converted to decimal in essentially the same manner as a binary number is converted to decimal:

$$\begin{aligned} 565_8 &= 5 \times 8^2 + 6 \times 8^1 + 5 \times 8^0 \\ &= 5 \times 64 + 6 \times 8 + 5 = 373_d \end{aligned}$$

Table 6.2 Octal and Hexadecimal Number Systems

Decimal (radix = 10)	Binary (radix = 2)	Octal (radix = 8)	Hexadecimal (radix = 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

This result is identical to that obtained earlier and is probably somewhat easier to obtain, since fewer manipulations are required. The main advantage of octal is, however, that the number 565 is more convenient to work with than 101110101.

Conversion from binary to hexadecimal is similar to conversion from binary to octal except that the digits are examined in groups of four rather than in groups of three. For 101110101₂ the conversion proceeds as follows:

1 0111 0101 (binary)

1 7 5 (hexadecimal)

Conversion from hexadecimal to decimal is analogous to conversion of binary or octal to decimal (subscript *h* designates a hexadecimal number):

$$175_h = 1 \times 16^2 + 7 \times 16^1 + 5 \times 16^0$$

$$= 1 \times 256 + 7 \times 16 + 5 = 373_d$$

which is the same result obtained earlier. Again, 175_h is easier to remember than 1 0111 0101₂.

In working with fractions, the same approach is used except that the digits in the fractional part are examined in groups of three or four starting with the binary point, as illustrated by the following examples for converting 101011.01001₂ to octal and hexadecimal:

Octal	Hexadecimal
101 011•010 010 ₂	10 1011 • 0100 1000 ₂
5 3 • 2 2 ₈	2 B • 4 8 _h

These are converted to decimal as follows:

$$\begin{aligned} 53.22_8 &= 5 \times 8^1 + 3 \times 8^0 + 2 \times 8^{-1} + 2 \times 8^{-2} \\ &= 5 \times 8 + 3 + 2 \times 0.125 + 2 \times 0.015625 \\ &= 43.28125_d \\ 2B.48_h &= 2 \times 16^1 + 11 \times 16^0 + 4 \times 16^{-1} + 8 \times 16^{-2} \\ &= 2 \times 16 + 11 + 4 \times 0.0625 + 8 \times 0.00390625 \\ &= 43.28125_d \end{aligned}$$

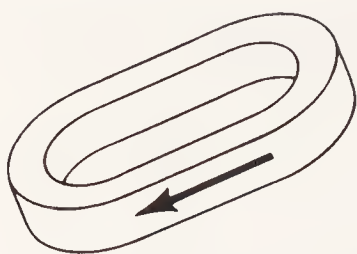
Converting 101011.01001₂ directly gives the same result.

6.2. STORAGE OF INFORMATION
IN A DIGITAL COMPUTER

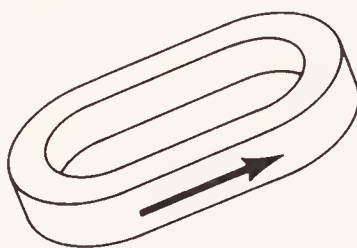
The storage unit on the digital computer consists of two-state devices capable of representing the two digits in the binary number system. A simple electric switch such as found on the wall in most modern homes is a possible candidate. The “on” position could represent one of the binary digits, say, “one,” and the “off” position could represent the other. But compared with the operational speed of modern computers, the common switch is far too slow to be considered. In the switch, electrical contact is made by positioning a mechanical element whose movement time is appreciable in terms of the time scale on which computers now operate.

In the 1960s, the industry standard was to use small ferrite (iron) rings or cores whose magnetic polarity could be switched, as illustrated in Figure 6.1. Again, one polarity was assigned the binary digit 1 and the other assigned the binary digit 0. The magnetic polarity of these devices could be switched very quickly. In the late 1960s, this switching time had been reduced to 1 μsec (microsecond, or one millionth of a second) or less. Use of these ferrite cores led to memory units being commonly referred to as *core storage*.

As the development of computers continues, the use of cores to construct memory units is imposing some limitations. The time required to reverse the magnetic polarity of the cores decreases as the size of the cores decreases. But since at least three wires must be threaded through each core, this places a lower limit on the size of cores. Currently it appears that cores with switching times much less than 1 μsec cannot be built on a commercial basis. This has led to a substantial effort to develop an economical memory unit to replace core storage



a. Clockwise polarization



b. Counterclockwise polarization

Figure 6-1 Magnetic cores

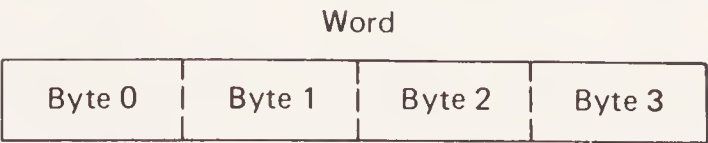
devices and as of this writing at least one machine with a solid-state memory is commercially in production.

Although the mechanism by which memory is implemented is extremely important in the design of the computer, the programmer is generally more interested in the manner in which the information is transmitted to and from memory. Of course, the smallest unit of information that can be stored is the binary digit or *bit* (contraction of *binary digit*). Very few computers, however, permit the programmer to store and retrieve one bit at a time.

In many machines, the *word* is the basic unit of information that is transmitted to and from memory. Depending upon the computer, the number of bits in a word ranges from about twelve up to about sixty, with thirty-two or thirty-six bits being near the norm. Each word in memory is given a unique address, and such machines are said to be *word-addressable*. On these machines, the term *storage-location* almost invariably refers to a word. In transmitting to and from memory, all bits in the word are transmitted simultaneously, or *in parallel*.

As we shall see later in this chapter, six to eight bits are sufficient to represent each character recognizable by the computer. Therefore, in many problems information is manipulated in fewer bits than usually comprise a word. This led to the design of byte-addressable machines in which the *byte* is the basic unit of information transmitted to or from memory. The number of bits in a byte equals the number of bits, generally six or eight, required to represent the characters recognized by the machine. Most byte-addressable machines, however, have tended to use eight-bit bytes. In these machines, the address refers to a unique byte in memory.

Although the byte is the basic unit of information in a byte-addressable machine, the word continues to carry significance on many of these computers. Beginning with the IBM 360 series, the general practice has been for a word to be composed of four 8-bit bytes, giving a 32-bit word:



The address of the word is the address of the leftmost byte. The restriction is made, however, that the address of the leftmost byte is a multiple of 4; or, in other words, if the address of the leftmost byte is expressed in binary, the last two digits are zeroes.

In addition to the word, the halfword and doubleword also have significance. The halfword consists of two bytes with the restriction that the address of the leftmost byte must be a multiple of 2. The doubleword consists of eight bytes with the restriction that the address of the leftmost byte must be a multiple of 8. These various configurations are illustrated in Figure 6-2.

On these machines, the term “storage location” is somewhat nebulous. When the term is used in context of character manipulation, a storage location is probably a byte. But since numerical data is generally stored in words, in this context the term storage location probably designates a word.

Regardless of whether the machine is word- or byte-addressable, memory is generally provided on units in which the number of words or bytes is an integral power of 2 or multiple thereof. For example, suppose we use ten binary digits to represent an address. Then $0000000000_2 = 0_d$ is the first address and $1111111111_2 = 1023_d$ is the last address, which gives a total of $1024_d = 2^{10}$

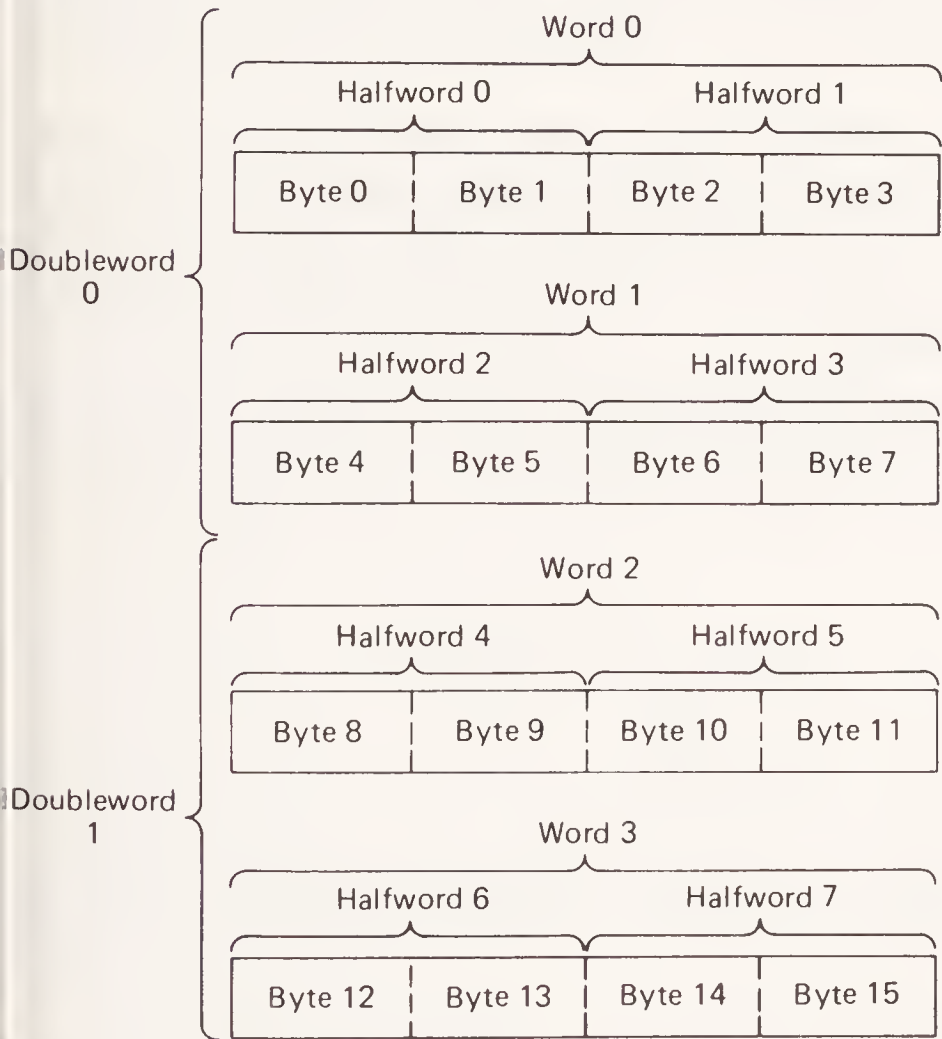


Figure 6-2 Arrangement of memory in a byte-oriented machine

210 unique addresses. A unit of 1,024 bytes is often referred to as a *kilobyte* or *kiloword* and abbreviated K. For example, a memory with 16K words would contain $16 \times 1,024 = 16,384$ words. A memory with 512K bytes would contain $512 \times 1,024 = 524,288$ bytes.

6.3. BINARY REPRESENTATION OF NUMERICAL DATA

Three different formats are currently in common use for storage of numerical data: fixed point (integer), floating point (real), and binary coded decimal. In this section, we shall examine the fixed-point and floating-point representations of numerical data, reserving binary coded decimal for a later section. In our examples, a thirty-two-bit word length will be generally assumed.

Fixed Point

Representation of integer numbers is readily accomplished by simple conversion to binary. Suppose we are using a machine whose word length is sixteen bits. Using the leftmost bit as the sign, the largest decimal integer that we can represent is $2^{15} - 1$, or 32,767. Using a thirty-two-bit word length and reserving the leftmost bit for the sign, the largest integer we can store is 2,147,483,647. This type of representation is referred to as *fixed point*, or *integer*, and is depicted in Figure 6-3a for a machine with a thirty-two-bit word length. Storage of the decimal number 473, or $1\ 1101\ 1001_2$, is illustrated in Figure 6-3b.

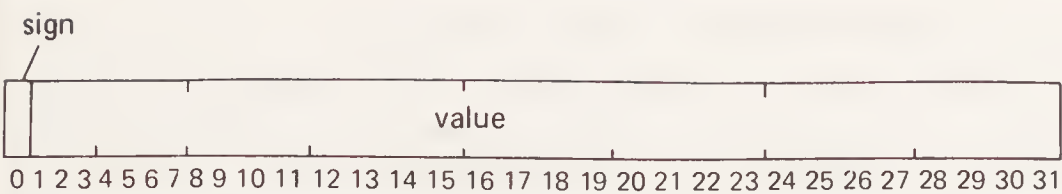
Negative numbers can be stored in one of three possible ways. The most straightforward is when the magnitude is stored directly and the sign bit turned “on,” that is, set equal to 1. Figure 6.3c illustrates this approach to storing -473. A more common approach, however, is to use either the radix complement or the radix-minus-one complement.

Before illustrating these complements in the binary system, suppose we would like to represent -473 as a six-digit complement in the decimal system. To obtain the radix complement (namely, the tens complement), -473 is subtracted from 10^6 :

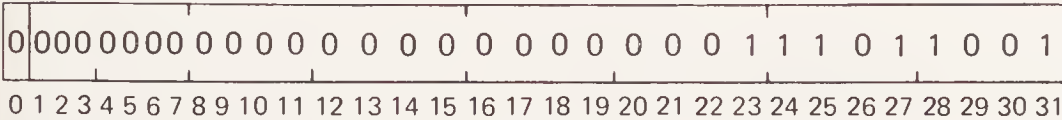
$$\begin{array}{r} 1000000 \\ - 473 \\ \hline 999527 \end{array}$$

To obtain the radix-minus-one complement, or nines complement, -473 is subtracted from $10^6 - 1$:

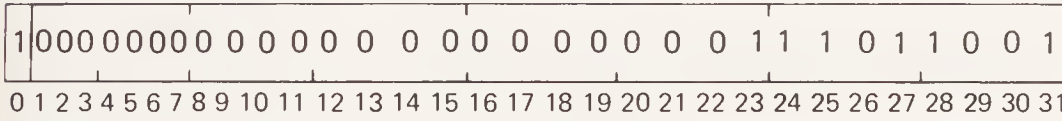
$$\begin{array}{r} 999999 \\ - 473 \\ \hline 999526 \end{array}$$



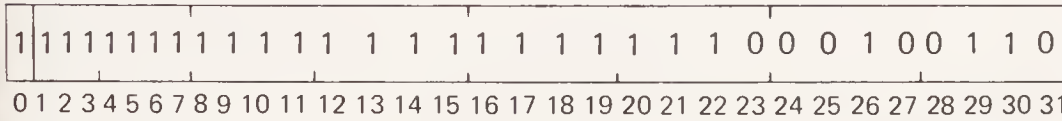
a. Format for fixed-point storage



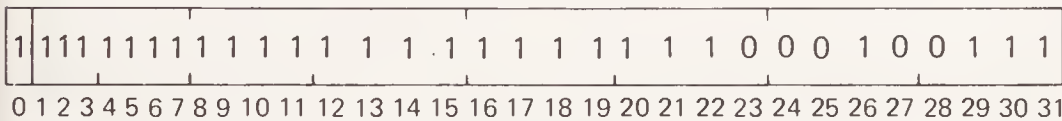
b. Storage of 473 in fixed-point format



c. Storage of -473 as magnitude-plus-sign



d. Storage of -473 in ones complement



e. Storage of -473 in twos complement

As illustrated by this example, the radix complement can be obtained by simply adding one to the radix-minus-one complement, a relationship that is very useful in the binary system.

In the binary number system, the radix complement is the twos complement and the radix-minus-one complement is the ones complement. Suppose we begin by representing $-473_d = -1\ 1101\ 1001_2$ as a sixteen-bit ones complement:

1111 1111 1111 1111

-1 1101 1001

1111 1110 0010 0110

(ones complement)

Note that to form the ones complement, each of the bits in the original number is simply changed to the other state; that is, all 1s are changed to 0s, and vice versa. To obtain the twos complement, we simply add 1 to the ones complement:

1111 1110 0010 0110

+ 1

1111 1110 0010 0111

(twos complement)

212 The twos complement is generally preferred. (See Exercise 6-2.)

Although use of the complement facilitates design of the arithmetic unit by enabling subtraction to be accomplished by adding (see Appendix E), the main advantage is that the operation of adding bits from the left is more direct. For example, if -473 is expressed as a sign-plus-magnitude number in sixteen bits, we obtain

$$1000\ 0001\ 1101\ 1001$$

To extend this number to twenty-four-bits, the sign bit must be given special treatment:

$$1000\ 0000\ 0000\ 0001\ 1101\ 1001$$

To extend the twos complement representation of -473 , namely,

$$1111\ 1110\ 0010\ 0111$$

to twenty-four-bits, we simply add 1s to the left to obtain

$$1111\ 1111\ 1111\ 1110\ 0010\ 0111$$

Note that no bit of the original sixteen-bit number is changed when the complement is used, whereas the sign bit of the original sixteen-bit number must be changed when working without complements. Thus, use of complements tends to remove some of the “special” characteristics that would be associated with the sign bit if complements were not used.

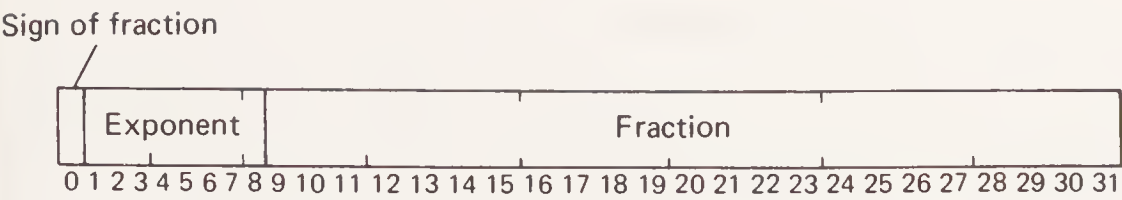
Upon encounter with an integer number consisting of more digits than there are bits in a word, a condition known as *integer overflow* is said to occur. On many computers, the result is that the thirty-two least significant bits are retained and the excess most significant bits are lost. Some machines generate error messages upon such an occurrence; others do not.

Since the decimal point (or binary point) in the integer notation is always assumed to be immediately to the right of the rightmost bit, two problems occur in the use of integer notation. First, fractional parts cannot be directly stored, which in itself is a serious limitation. Second, extremely large numbers, for example, 6.23×10^{23} , cannot be stored directly.

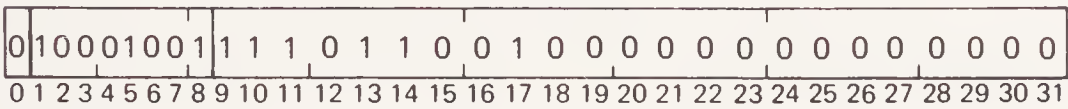
Floating Point

To circumvent this difficulty, the floating-point notation is used. Decimal numbers can be represented in the scientific or exponential notation, for example,

$$473 = 0.473 \times 10^3$$



a. Format of binary floating-point storage



b. Storage of 473 in binary floating-point format

Figure 6-4 Floating-point format

This representation includes a fraction, 0.473, and an exponent, 3. If the fraction is less than 1.0 but greater than or equal to 0.1 (that is, first digit to the right of the decimal point is nonzero), the fraction is said to be *normalized*.

For binary numbers, a similar floating-point notation can be used. Recall that the decimal number 473 is represented in binary as 1 1101 1001₂. Floating the binary point gives the representation

$$0.1110\ 1100\ 1_2 \times 2^9$$

The value of the exponent is simply the number of places the binary point was shifted to the left to obtain a normalized fraction. The decimal representation is

$$0.92382815 \times 2^9 = \frac{473}{512} \times 2^9 = 473$$

To store a number in the binary floating-point notation requires that both the fraction and the exponent be stored. This is usually done in a single word as illustrated in Figure 6-4a. The leftmost bit is generally reserved for the sign of the fraction, the next group of bits for the exponent, and the remainder of the word is used to store the fraction. The complement notation is *not* used for negative fractions.

Although a sign bit is reserved in Figure 6-3a for the fraction, no sign bit is specifically designated for the exponent. This is because the number 1000 0000₂, or decimal 128, is automatically added to the exponent prior to storage to obtain a quantity known as the *characteristic*, which is stored instead of the true exponent. The value of the exponent is therefore obtained by subtracting 128 from the quantity in storage. That is, the quantity 1000 1001₂ is decimal 137. Subtracting 128 yields 9 as the value of the exponent. The quantity 0101 0010₂ is decimal 82, or an exponent value of -46. The quantity 1000 0000₂ corresponds to a zero exponent.

The storage of the number 473 = 0.1110 1100 1₂ × 2⁹ in binary floating point is illustrated in Figure 6.4b.

The largest exponent that can be stored is 1111 1111₂, which corresponds to a value 255 - 128 = 127. The largest number that can be stored is

$2^{127} \cong 10^{38}$

The value zero corresponds to all bit positions in the word being zero. Whenever the fraction becomes zero, the exponent bits are also set equal to zero.

Some machines, notably the IBM System/360, use a hexadecimal floating-point notation as opposed to a true binary floating-point notation. In this case, the number is expressed as a fraction multiplied by a power of 16. The number 473 is represented as follows:

$$\begin{aligned} 1\ 1101\ 1001 &= 1\ 1101.1001_2 \times 16 \\ &= 1.1101\ 1001_2 \times 16^2 \\ &= 0.0001\ 1101\ 1001_2 \times 16^3 \\ &= 0.1D9_h \times 16^3 \end{aligned}$$

In this case, the fraction is normalized whenever the first hexadecimal digit is nonzero, which as illustrated above does not necessarily mean that the first binary digit is nonzero.

The layout of the floating-point word for the System/360 is illustrated in Figure 6-5a. One bit fewer is reserved for the exponent as compared with the binary format in Figure 6-4a, but we shall see that we are actually able to store larger exponents in the hexadecimal format. Before storing the exponent, the number 100 0000₂ or 40_h, or decimal 64, is added to give the characteristic. Storage of decimal 473 in hexadecimal floating point is illustrated in Figure 6-5b.

Using hexadecimal floating point, the largest exponent that can be stored is 111 1111₂ - 100 0000₂ = 127 - 64 = 63. The largest number that can be stored is

$16^{63} \cong 10^{76}$

as compared with 10³⁸ for the binary floating-point format in Figure 6-4a.

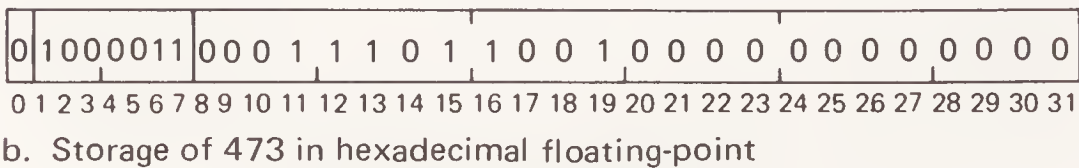
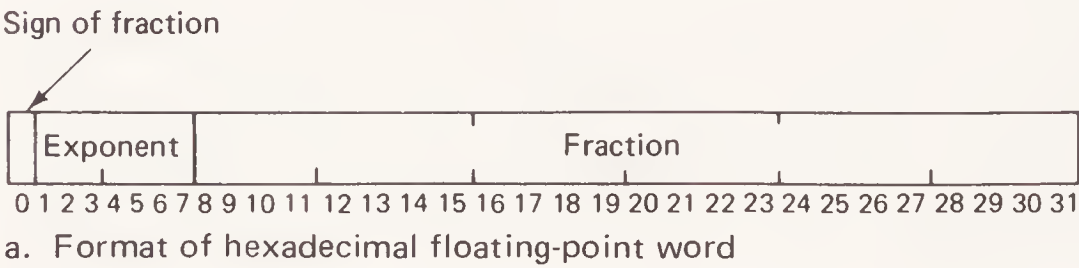
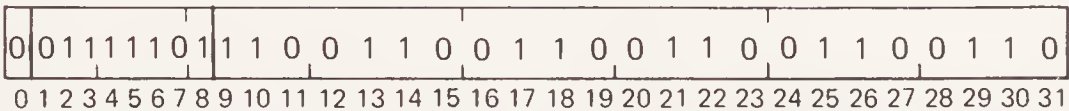


Figure 6-5 Hexadecimal floating-point

One property of floating-point representations that causes some difficulties is the inexact representation of some decimal numbers. For example, the number 0.1 is represented in binary as follows:

$$0.1 = 0.0001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 100 \dots$$
$$0.1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100 \dots \times 2^{-3}$$

Using the binary format in Figure 6-4a, this number is stored as follows:



Converting back to decimal gives 0.0999999930295 . . . , which is not exactly 0.1.

For the binary floating-point format given in Figure 6-4, we concluded that the largest exponent that could be stored was $2^{127} \cong 10^{38}$. The smallest that could be stored was $2^{-128} \cong 10^{-38}$. It is certainly plausible that somewhere in the program the following multiplication could be requested:

$$(0.19 \times 10^{31}) \cdot (0.20 \times 10^{10})$$

The exponent in the result, 0.38×10^{41} , would be too large for the computer to store, resulting in what is called a *floating-point overflow*.

Alternatively, the product of the two numbers

$$(0.19 \times 10^{-31}) \cdot (0.20 \times 10^{-10}) = 0.38 \times 10^{-41}$$

has an exponent that is too small (that is, too large negatively) for the computer to store, resulting in a *floating-point underflow*.

In most systems, messages are generated whenever either of these occur. Although very large numbers (that is, in excess of 10^{38}) are encountered in a few problems, floating-point overflows generally indicate that the program logic is not correct, that there is an error in a parameter, or that some other erroneous condition exists.

Floating-point underflows sometimes result from evaluating expressions that should give a zero result, but owing to roundoff or other problems the result is very small but not exactly zero. In these cases, the result can usually be set equal to the true zero and the computations continued, unless the result is later multiplied by an extremely large number. That is, consider the following operations:

$$(10^{-10} \times 10^{-30}) \times 10^{25} = 10^{-15}$$

Of course, multiplying 10^{-10} by 10^{-30} would give a floating-point underflow

under the format in Figure 6-4. If this were set equal to zero as proposed above and then the result multiplied by 10^{25} , the product is zero, not 10^{-15} .

Although we have discussed floating-point underflows and overflows only in terms of multiplication, they can also arise from addition, subtraction, and division. In fact, overflows often arise from dividing by a number that is nearly zero.

6.4. STORAGE OF CHARACTERS

Representation of characters is via a character code in which each character is assigned a unique combination of bits. Table 6-3 illustrates the three character codes in common use. The Binary Coded Decimal Interchange code (BCDIC) was widely used until the advent of the IBM System/360, which used the Extended Binary Coded Decimal Interchange Code (EBCDIC). These two are very similar except that EBCDIC contains more characters. BCDIC cannot, however, be considered a subset of EBCDIC because the code of a few characters, notably +, (,), and =, in EBCDIC were changed from their code in BCDIC. A third common code is ASCII (American Standard Code for Information Interchange), which differs significantly from both BCDIC and EBCDIC.

Using a six-bit character code, the first character is assigned $000\ 000_2 = 0_d$ and the last character is assigned $111\ 111_2 = 63_d$, giving a maximum of $64_d = 2^6$ unique characters. This is sufficient for the twenty-six upper-case letters, the ten digits, and twenty-eight special characters such as the comma and the period. In fact, a total of forty-eight characters has proven sufficient for most situations. If, however, distinction is to be made between upper-case and lower-case letters, sixty-four characters are not sufficient. With twenty-six upper-case characters, twenty-six lower-case characters, and the ten digits, a maximum of two special characters could be represented by a six-bit character code. Use of an eight-bit character code permits representation of a total of 256 different characters, which is generally adequate.

Since the number of bits in a six-bit character code is an integer multiple of three, the octal numbering system is a convenient shorthand notation. The hexadecimal number system is likewise convenient when an eight-bit character code is used.

On byte-addressable machines, individual characters can be stored and retrieved from memory, which certainly facilitates character manipulation on these machines. To achieve efficient utilization of memory on word-addressable machines, several characters must be stored in each word, which entails packing and unpacking operations. For example, on a machine with a thirty-six-bit word and a six-bit character code, six characters can be stored in each word. To obtain one of these characters, however, all thirty-six-bits are retrieved from memory and a sequence of shifting operations are performed to isolate the desired character.

6.5. DECIMAL NUMBERS

To achieve maximum computational efficiency, the fixed-point and floating-point representations of numbers described in Section 6.3 are used almost exclusively in scientific computations. But since these representations are so different from the representation of the digits that comprise the number, their use does not lead to maximum efficiency in problems in which input/output speed is of utmost importance. These applications have historically appeared most frequently in business-data processing, which has resulted in the binary-coded decimal representation being favored for most business problems that involve little computation but substantial input/output.

The basic concept of the binary-coded decimal representation is simply to convert each digit to binary. Since the digits 0 through 9 must be converted, four bits are required for each decimal digit. For example, the decimal number 493 would be represented as follows:

4 9 3
0100 1001 0011 (binary coded decimal)

This is not identical to the true binary representation, $1\ 1110\ 1101_2$.

Each of the representations of the digits in the eight-bit character codes in Table 6-3 consists of two fields: a four-bit zone field followed by a four-bit digit field. In EBCDIC, the zone field is 1111 in binary, or F in hexadecimal, for each of the ten digits; in ASCII, the zone field is 0101 in binary, or 5 in hexadecimal. For example, if the number 493 is represented in EBCDIC, the result is

1111 0100 1111 1001 1111 0011

In other words, the representation is of the following format:

Zone	Digit	Zone	Digit	Zone	Digit
------	-------	------	-------	------	-------

If the zone fields are removed, the result is the number represented in binary-coded decimal. That is, removing the zone fields from the above EBCDIC representation of 493 gives

0100 1001 0011

which is the binary-coded decimal representation of 493_d .

Up to this point, we have said nothing about the sign. In the *zoned format*, the zone field of the rightmost digit is replaced by a sign field, giving

Zone	Digit	Zone	Digit		Zone	Digit	Sign	Digit
------	-------	------	-------	-------	------	-------	------	-------

Table 6-3 Character Codes

Character	BCDIC ^a	EBCDIC	ASCII
0	001 010	1111 0000	0011 0000
1	000 001	1111 0001	0011 0001
2	000 010	1111 0010	0011 0010
3	000 011	1111 0011	0011 0011
4	000 100	1111 0100	0011 0100
5	000 101	1111 0101	0011 0101
6	000 110	1111 0110	0011 0110
7	000 111	1111 0111	0011 0111
8	001 000	1111 1000	0011 1000
9	001 001	1111 1001	0011 1001
A	110 001	1100 0001	0100 0001
B	110 010	1100 0010	0100 0010
C	110 011	1100 0011	0100 0011
D	110 100	1100 0100	0100 0100
E	110 101	1100 0101	0100 0101
F	110 110	1100 0110	0100 0110
G	110 111	1100 0111	0100 0111
H	111 000	1100 1000	0100 1000
I	111 001	1100 1001	0100 1001
J	100 001	1101 0001	0100 1010
K	100 010	1101 0010	0100 1011
L	100 011	1101 0011	0100 1100
M	100 100	1101 0100	0100 1101
N	100 101	1101 0101	0100 1110
O	100 110	1101 0110	0100 1111
P	100 111	1101 0111	0101 0000
Q	101 000	1101 1000	0101 0001
R	101 001	1101 1001	0101 0010
S	010 010	1110 0010	0101 0011
T	010 011	1110 0011	0101 0100
U	010 100	1110 0100	0101 0101
V	010 101	1110 0101	0101 0110
W	010 110	1110 0110	0101 0111
X	010 111	1110 0111	0101 1000
Y	011 000	1110 1000	0101 1001
Z	011 001	1110 1001	0101 1010
blank	010 000	0100 0000	0010 0000
=	001 011	0111 1110	0011 1101
+	110 000	0100 1110	0010 1011
—	100 000	0110 0000	0010 1101
*	101 100	0101 1100	0010 1010
/	010 001	0110 0001	0010 1111
(	011 100	0100 1101	0010 1000

^aThe BCDIC code given here is the six-bit code used on the CDC 6000 series computers.

Table 6-3 (Continued)

Character	BCDIC ^a	EBCDIC	ASCII
)	111 100	0101 1101	0010 1001
,	011 011	0110 1011	0010 1100
.	111 011	0100 1011	0010 1110
\$	101 011	0101 1011	0010 0100
'		0111 1101	0010 0111
<	111 010	0100 1100	0011 1100
>	101 111	0110 1110	0011 1110
:		0111 1101	0011 1010
;		0101 1110	0011 1011
"	111 111	0111 1010	0010 0010

In EBCDIC, the sign fields for plus and minus are 1100 and 1101, respectively. Many systems also consider a 1111 in the sign field to be plus, thereby making unsigned numbers positive.

While the zone fields are necessary in order to represent the characters, they are not necessary when the numbers are being manipulated internally. Instead, the following *packed* format leads to more efficient utilization of memory:



To obtain the packed representation from the zoned (or unpacked) representation, we need only to (a) remove all zone fields and (b) transpose the sign and digit fields in the rightmost byte. This is illustrated below for -493:

zoned 1111 0100 1111 1001 1100 0011

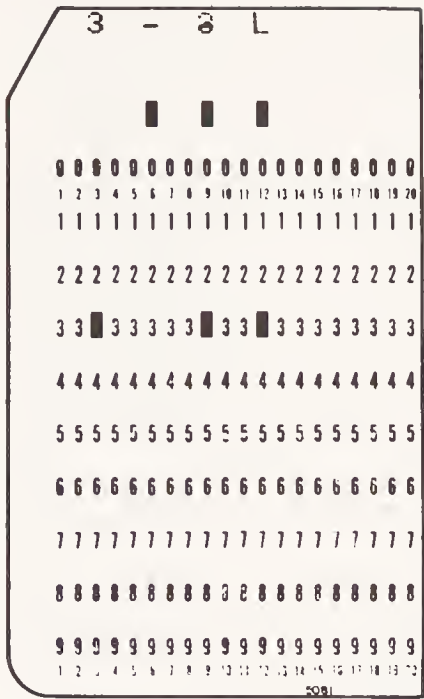
packed 0100 1001 0011 1100

Systems such as the IBM 360 have hardware instructions to “pack” and “unpack” decimal numbers.

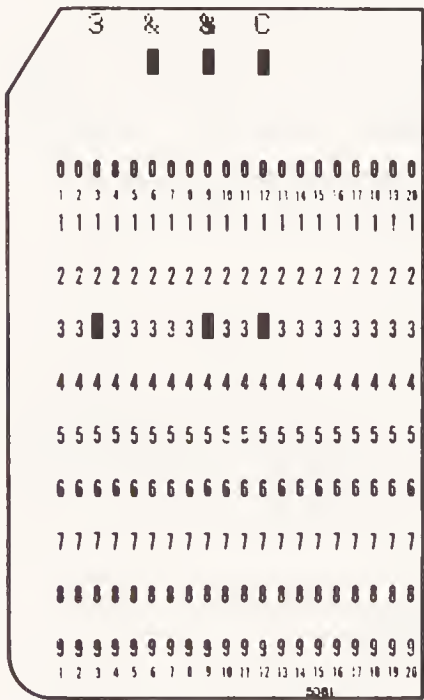
When programs written in programming languages such as Cobol read numerical data, the sign does not normally occupy a separate column. Instead, it is entered as an *overpunch* in the rightmost digit. For example, suppose the number -493 is to be entered. If the minus sign and the digit 3 are punched in the same column, as illustrated in Figure 6-6, the resulting combination of holes is identical to that of the character L. From Table 6-3, the EBCDIC code for L is

1101 0011

The left four bits form the sign field and indicate that the number is negative; the right four bits are the binary representation of the decimal number 3.



a. Overpunch (3 and -)



b. Overpunch (3 and &)

Figure 6-6 Use of overpunch for sign

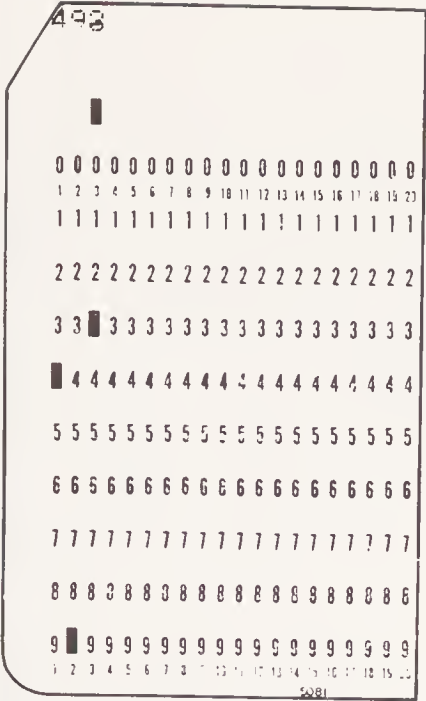
Figure 6-6 also illustrates the situation when the & symbol[†] is overpunched in the same column as the 3. This gives the character C whose EBCDIC code is

1100 0011

The lefthand four bits constitute the sign field and indicate a positive number, while the righthand four bits represent the decimal number 3.

Figure 6-7 summarizes the entry of +493, -493, and 493 unsigned.

[†]In the original BCD character set, the + and & symbols were equivalent in that they corresponded to the same hole punches on the card. When the character set was extended to EBCDIC, the + symbol was given a new punch configuration which necessitated that the & symbol instead of the + be used as the overpunch.



a. Entry of -493

Zoned format:

1111 0100 1111 1001 1101 0011
zone digit zone digit sign digit

Packed format:

0100 1001 0011 1101
digit digit digit sign



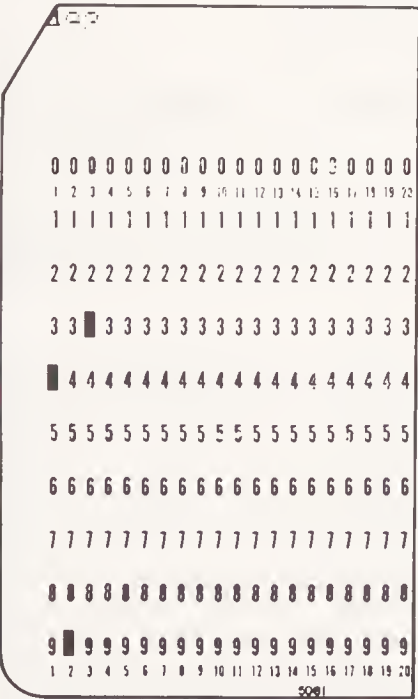
b. Entry of +493

Zoned format:

1111 0100 1111 1001 1100 0011
zone digit zone digit sign digit

Packed format:

0100 1001 0011 1100
digit digit digit sign



c. Entry of 493 unsigned

Zoned format:

1111 0100 1111 1001 1111 0011
zone digit zone digit sign digit

Packed format:

0100 1001 1001 1111
digit digit digit sign

Figure 6-7 Entry of signed numbers

6.6. MACHINE INSTRUCTIONS

Each computer is capable of performing a certain number of elementary operations that comprise the instruction set. Examples of these instructions include loading the contents of a location in memory into a register (called the *accumulator*) in the central processing unit, adding the contents of a storage location to the accumulator, and branching to another instruction depending upon the results of a certain operation. The total number of instructions varies from machine to machine, some having as few as twenty instructions and some having over two hundred.

Of course, the content of the instruction set varies widely even between machines having approximately the same number of total instructions. In addition, the instruction format, addressing scheme, number of registers, etc., also vary widely. Since our discussion here is to be introductory in nature, we shall begin with a very simple computer, one that could have existed a few years ago but is not very likely to be found today. Toward the end of this section, we shall introduce several extensions that are commonly found in today's computers.

In our simple machine, we shall assume that only three registers will be used. They are:

1. The *instruction address register* or *program counter* that contains the address of the *next* instruction to be executed.
2. An *instruction register* that contains the instruction that is currently being executed.
3. The accumulator.

In Section 6.8 we shall add an index register.

The processing of an instruction occurs in two phases:

1. The *fetch* phase proceeds as follows:
 - a. The contents of the word at the address currently contained in the program counter are loaded into the instruction register.
 - b. The address in the program counter is incremented by one.
2. The *execution* phase consists of performing whatever operations are indicated by the contents of the instruction register.

Upon completion of the execution phase, the fetch phase is re-entered, initiating another cycle.

In its simplest form, the format of an instruction consists of two fields. One of these is reserved for the operation code (or "op code" for short), which designates the operation to be performed. The second field designates the address of whatever data are to be used in executing the instruction. For example, suppose we want to load the contents of address 147 into the accumulator, which is done by an instruction usually referred to as "load word." The op code portion of the instruction contains the bit pattern that designates the load-word operation; the address portion contains the bit pattern corresponding to 147.

Suppose our simple computer has a sixteen-bit word-length. In addition, suppose we reserve four bits for the op code, leaving twelve bits for the address. Therefore, the maximum number of instructions is 2^4 , or 16; the largest address that can be referenced is $2^{12} - 1$, or 4095. For machines with a short word length, the designer is presented with a tradeoff: as he reserves more bits for the op code, the number of addresses he can reference from the address field decreases. Designers have devised ingenious ways to deal with this problem, some of which we shall examine shortly.

Table 6-4 gives a portion of the instruction set for our simple machine. We have intentionally omitted some of the more complex classes of instructions, notably those dealing with input/output operations. We have also omitted instructions for integer multiply, integer divide, floating-point operations, and decimal operations. However, these instructions are sufficient to illustrate how the computer functions.

Specifically, suppose we would like to move the contents of location 96 to location 117. Using the instruction set in Table 6-4, the steps would be as follows:

- 1. Place the contents of location 96 in the accumulator.
- 2. Store the contents of the accumulator into location 117.

The instructions are

LDW96

STW117

After these instructions are executed, location 96, location 117, and the accumulator contain the same information.

Next, suppose we would like to add the contents of location 206 to that of location 148, storing the result in location 186. The necessary instructions are

LDW148

ADD206

STW186

The contents of addresses 148 and 206 are unchanged.

For a more complex example, suppose you want to compute the fourteenth number in the Fibonacci sequence. As described in Section 3.3, the appropriate flowchart is as given in Figure 6-8. Suppose storage is allocated as follows:

Variable or constant	Location
NEXT	75
CURRENT	76
LAST	77
1	78
14	79
COUNTER	80

Table 6-4

Mnemonic	Binary	Decimal	Hexadecimal	Explanation (ADR = address in address field)
HLT	0000	0	0	“Halt”; the machine stops; address field is not used
LDW	0001	1	1	“Load word”; the contents of ADR are transferred to the accumulator
STW	0010	2	2	“Store word”; the contents of the accumulator are transferred to ADR
ADD	0011	3	3	“Add”; the contents of ADR are added to the accumulator, the result appearing in the accumulator; arithmetic is in the integer mode
SUB	0100	4	4	“Subtract”; the contents of ADR are subtracted from the accumulator, the result appearing in the accumulator; arithmetic is in the integer mode
B	0101	5	5	“Branch”; the next instruction to be executed is the one whose address is ADR; execution of this instruction requires that ADR be placed in the program counter
BM	0110	6	6	“Branch on minus”; if the value in the accumulator is negative, the next instruction to be executed is the one at ADR; otherwise, execute the next instruction; execution of the BM instruction requires that ADR be placed in the program counter if the value in the accumulator is negative.
RSF	0111	7	7	“Right shift”; the contents of the accumulator are shifted one bit to the right
LSH	1000	8	8	“Left shift”; the contents of the accumulator are shifted one bit to the left

The appropriate instructions are given in Table 6-5. The sequence follows the flowchart in Figure 6-8 very closely until the decision block. To use the “branch

on minus” instruction, the decision “COUNTER < 14” is transposed to “COUNTER – 14 < 0.”

A fanatic for efficient programming would notice that one instruction in the sequence in Table 6-5 can be eliminated by rearranging the order of the blocks in Figure 6-8. Specifically, the block

```
COUNTER ← COUNTER + 1
```

in Figure 6-8 can be shifted down to just prior to the decision block. After executing the instructions for this block, the accumulator contains COUNTER, which permits 14 to be subtracted directly. The revised instruction sequence is shown in Table 6-6.

To illustrate the influence of the instruction set on programming efficiency, consider programming the statement

```
COUNTER ← COUNTER + 1
```

This requires three instructions plus a storage location, specifically, location 78, to store the number 1.

Suppose we use a machine whose instruction set contains the “Load Immediate” (LDI) instruction, which causes the contents of the address field of the instruction to be transferred directly to the accumulator. In effect, we are using the address field to store data. In this case, the appropriate instructions are

```
LDI      1
ADD      80
STW      80
```

Three instructions are still required, but no separate location is needed to store the constant 1.

Next suppose we have a machine that executes an “increase by one” (INC) instruction, which causes the contents of the address appearing in the address field to be increased by one. Now COUNTER can be incremented by 1 using the single instruction

```
INC      80
```

Incidentally, this instruction is actually included in the instruction set of several recent machines.

6.7. USE OF MULTIPLE ACCUMULATORS

In the early 1960s, practically every computer was designed with a single accumulator. In many applications, the single accumulator became a bottleneck in the operation, entailing an excessive number of load-word and store-word instructions in order that the single accumulator could be used in performing the necessary operations. Provision of multiple accumulators was conceptually the

Table 6-5 Instruction Sequence Corresponding to the Flowchart in Figure 6-8

Address of instruction	Operation	Address field	
25	LDW	76	} NEXT ← CURRENT + LAST
26	ADD	77	
27	STW	75	
28	LDW	80	} COUNTER ← COUNTER + 1
29	ADD	78	
30	STW	80	
31	LDW	76	} LAST ← CURRENT
32	STW	77	
33	LDW	75	} CURRENT ← NEXT
34	STW	76	
35	LDW	80	} COUNTER ← 14
36	SUB	79	
37	BM	25	

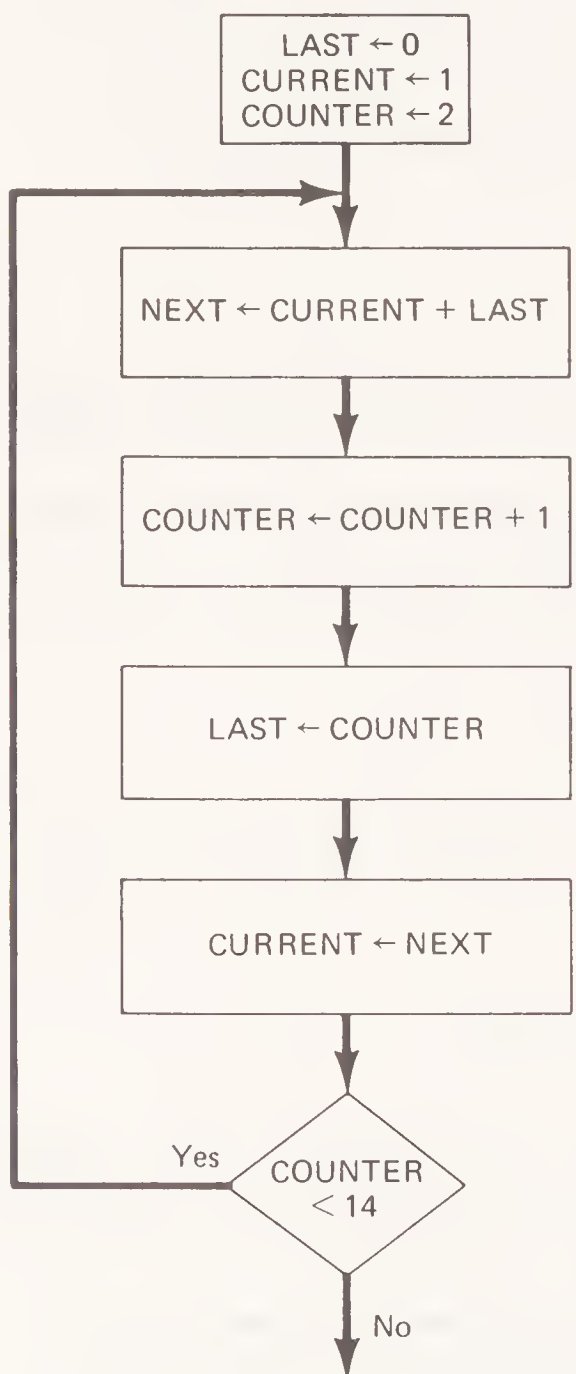


Figure 6-8 Computation of fourteenth entry in the Fibonacci sequence

Table 6-6 Improved Instruction Set over Figure 6-9

Address of instruction	Operation	Address field	
25	LDW	76	} NEXT ← CURRENT + LAST
26	ADD	77	
27	STW	75	
28	LDW	76	} LAST ← CURRENT
29	STW	77	
30	LDW	75	
31	STW	76	} CURRENT ← NEXT
32	LDW	80	
33	ADD	78	
34	STW	80	} COUNTER ← COUNTER + 1
35	SUB	79	
36	BM	25	

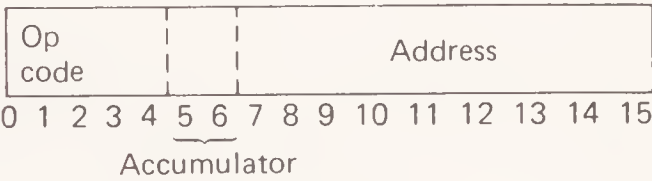
simplest answer, the only drawback being a more complex hardware design problem.

For example, suppose our computer has four accumulators. The first impact is that another field must be added to the instruction format. For example, in a single-accumulator machine, the instruction

LDW 2607

instructs the machine to load the contents of the storage location at address 2607 into *the* accumulator. On a machine with four accumulators, however, this instruction is ambiguous.

To remove this ambiguity, another field, the register field, must be added to the instruction format. Since two bits are sufficient to designate one of four accumulators, a possible instruction format for a sixteen-bit machine could be:



The instruction to load the contents of the storage location at address 2607 into accumulator 2 would be:

LDW,2 2067

All instructions except the halt, unconditional branch, and branch on minus in Table 6-4 should be modified accordingly.

The existence of multiple accumulators has a special significance to conditional branch instructions. While the above modification could be made on the “branch-on-minus” instruction, another alternative is generally chosen. Machines which utilize multiple accumulators usually evaluate conditional branches using

228 a special set of bits known as the *condition codes*. These bits are generally set as a result of the load-word instruction (LDW) and all arithmetic instructions (ADD and SUB in Table 6-4). A two-bit condition code might be set as follows:

- 00 indicates a zero result
- 01 indicates a negative result
- 10 indicates a positive result
- 11 indicates an error condition such as overflow

For example, if an ADD instruction added 276 to -421, the condition codes would be set to 10 during the execution of the instruction regardless of which accumulator is used in the ADD instruction.

In addition to “branch-on-minus” (BM), the “branch-on-zero” (BZ), and “branch-on-positive” (BP) instructions are also recognized.

To enable our computer to perform accumulator-to-accumulator operations, we shall assume that addresses 0, 1, 2, and 3 refer to the accumulators and *not* to storage locations in memory. For example, the instruction

```
LDW,2      3
```

copies the contents of accumulator 3 into accumulator 2. This instruction is equivalent to

```
STW,3      2
```

The add and subtract instructions can be treated in the same fashion.

The alternative to this approach is to implement special instructions for accumulator-to-accumulator operations, which was done on the IBM/360 series.

One of the advantages of a multiple-accumulator machine over a single-accumulator machine is that a given problem can generally be solved with fewer instructions on the former. For example, let’s again consider computation of the fourteenth entry in the Fibonacci sequence. The use of the four accumulators is as follows:

Accumulator	Use	Initial value
0	Counter	14
1	LAST	0
2	CURRENT	1
3	NEXT	immaterial

The only memory storage location used will be the one at address 78, which is assumed to contain the constant 1.

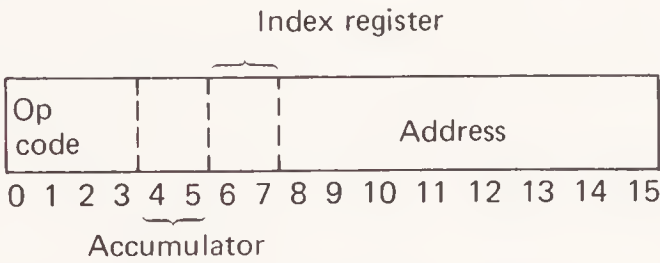
The following instructions will accomplish the desired task:

Address	Op code, accumulator	Address	Explanation
25	LDW,3	1	NEXT← CURRENT + LAST
26	ADD,3	2	
27	LDW,1	2	LAST← CURRENT
28	LDW,2	3	CURRENT← NEXT
29	SUB,0	78	Counter
30	BP	25	

Only six instructions are required as compared with twelve in Table 6-6. We have also implemented the counter more efficiently by initializing the counter at 14 and “down-counting,” that is, subtracting one, until the counter reaches zero. This improvement could be made in Table 6-6 to improve its efficiency also.

6.8. INDEXED ADDRESSING

The features of the simple computer described in the previous section would be sufficient to implement virtually all of the programming concepts described in Chapters 2 and 3 of this text. However, in order to implement the equivalent of the subscripted-variable concept, an *index register* must be added. This enables the machine to perform *indexed addressing*, whereby the contents of an index register are added to that of the address field to determine the actual address to be used. Although machines with sixteen index registers are common, our simple machine shall have only three index registers. To designate the appropriate index register, two bits in the instruction format must be reserved for this purpose. Using a sixteen-bit word, the instruction format could be as follows:



The bit pattern in bits 4 and 5 is interpreted as follows:

- 00 Use no index register
- 01 Use index register 1
- 10 Use index register 2
- 11 Use index register 3

In virtually all previous-generation computers and on a few manufactured

today, the index registers were implemented as separate registers that served this one purpose. This necessitated a set of special instructions such as “load index register,” “store index register,” etc., to manipulate the contents of these registers. The current trend, however, is toward machines with multiple accumulators, some of which can be used as index registers. In fact, when the accumulators can be used as index registers, they are often referred to as *general-purpose registers* or simply as *registers*.

In this section, we shall assume our machine has four registers that can function as accumulators and three that can function as index registers. Since a bit pattern of 00 in the index register field indicates no indexing, register 0 cannot be used as an index register.

To indicate indexing, we shall append the index register to the address, separating by a comma. For example, the instruction

```
ADD,2      1416,3
```

is interpreted as “add the contents of the storage location at address [1416 plus the contents of register 3] to the contents of register 2.”

To illustrate the use of indexed addressing, suppose we would like to compute the sum of the numbers in locations 1207 through 1214. If this area were reserved for a single-subscripted array A, the appropriate flowchart is given in Figure 6-9. In order to achieve a closer parallel with indexed addressing, we have intentionally assumed that A_0 is stored in location 1207 and that A_7 is stored in location 1214.

Suppose registers 0 and 1 are used as the counterparts to SUM and J,

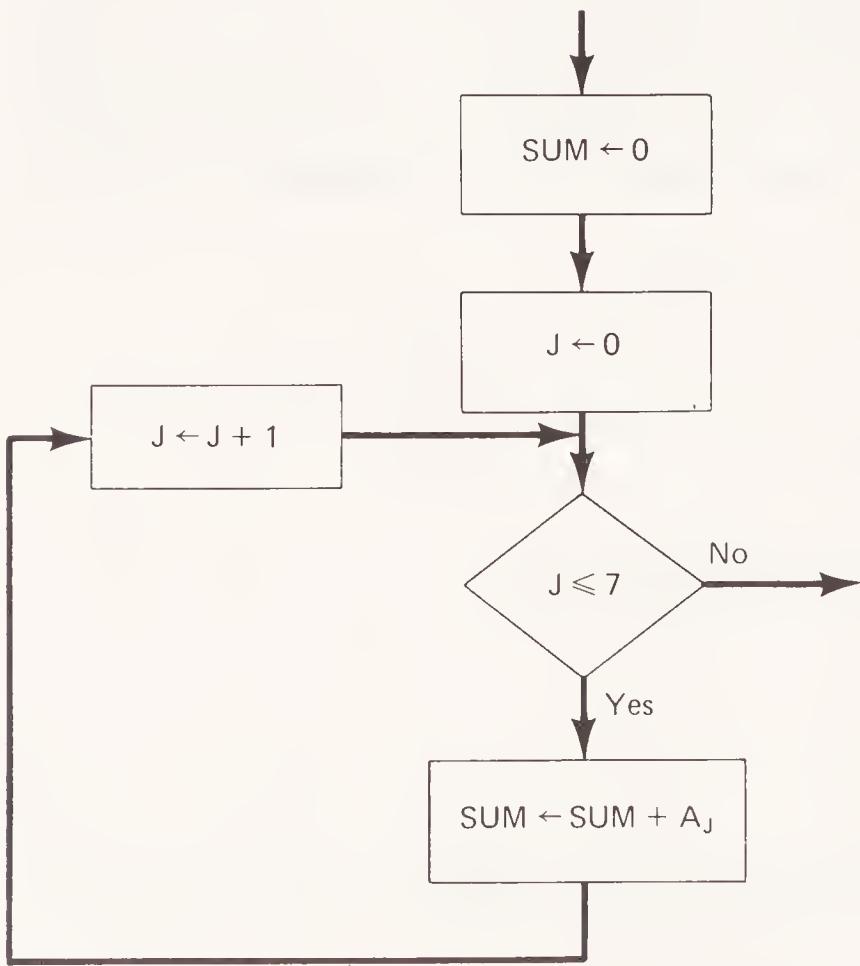


Figure 6-9 Flowchart for computing sum of eight numbers using subscripted variables

respectively, in Figure 6-9. Furthermore, suppose both of these registers are initialized to zero. Locations 78 and 79 in memory are assumed to contain the constants 1 and 8, respectively. Using indexed addressing, the instructions to accomplish the equivalent of the operations in Figure 6-9 are as follows:

Location	Op code, register	Address, index register	Explanation
25	ADD, 0	1207, 1	$SUM \leftarrow SUM + A_J$
26	ADD, 1	78	$J \leftarrow J + 1$
27	LDW, 2	79	} $J \leq 7 ?$
28	SUB, 2	1	
29	BM	25	

Alternatively, the SUM could be computed “backward”; that is, starting with location 1214, with the following instructions (register 1 initialized to 7):

Location	Op code, register	Address, index register	Explanation
25	ADD, 0	1207, 1	$SUM \leftarrow SUM + A_J$
26	SUB, 1	78	} COUNTER
27	BP	25	
28	BZ	25	

Either of these approaches leads to the correct answer, with the latter being slightly more efficient.

6.9. ASSEMBLY-LANGUAGE PROGRAMMING

The lowest level programming that is generally done is at the assembly-language level. In fact, the programs in Tables 6-5 and 6-6 are almost in assembly language except that in assembly language variable names are generally substituted for the addresses. That is, the assembler is responsible for allocation of storage locations.

Those interested in assembly-language programming should consult the examples in Appendix A. In this Appendix, a subset of the IBM System/360 assembly language and a subset of the XDS Sigma series assembly language are used to program simple problems. Students using either of these systems can program problems of their own using these instructions. Similar subsets can be developed for machines produced by other manufacturers.

EXERCISES

6-1. The information stored in the thirty-two-bit word in Figure 6-3b can be expressed in hexadecimal notation as 00001D9. Represent the information in the following figures in hexadecimal:

- a. 6-3c
- b. 6-3d
- c. 6-3e
- d. 6-4b
- e. 6-5b

6-2. Although the use of ones complement as opposed to twos complement notation leads to some advantages, there are also some disadvantages. To illustrate one disadvantage, determine the following:

- a. The ones complement representation of $+0$
- b. The ones complement representation of -0
- c. The twos complement representation of $+0$
- d. The twos complement representation of -0

In the ones complement notation $+0$ is not equal to -0 , a situation that does not exist for twos complement notation.

6-3. A machine has a memory capacity of 24K thirty-two-bit words. How many words and how many bytes are in this memory?

6-4. Determine the decimal equivalent of the following numbers (truncate decimal fractions after ten digits):

- a. 10110.101_2
- b. 1011101.10011_2
- c. 347.25_8
- d. 12.4276_8
- e. $A4.12C_h$
- f. $E2C.1F_h$

6-5. Represent the number $E2C.1FA4_h$ as follows:

- a. In the integer representation in Figure 6.3a.
- b. In the binary floating-point format of Figure 6.4a.
- c. In the hexadecimal floating-point format of Figure 6.5a.

6-6. Represent the number 0.0002746120465_8 as follows:

- a. In the integer representation in Figure 6.3a.
- b. In the binary floating-point format of Figure 6.4a.
- c. In the hexadecimal floating-point format of Figure 6.5a.

6-7. The characters GIRL are entered in the EBCDIC character code and stored in a thirty-two-bit word in memory. What numerical value is obtained if this information is then treated in the following manner (truncate decimal fractions after ten digits):

- a. An integer number according to the format in Figure 6.3. Negative numbers are represented in two's complement notation.
- b. A binary floating-point number according to the format in Figure 6.4.

c. A hexadecimal floating-point number according to the format in Figure 6.5.

6-8. Same as Exercise 6-8 except that the characters are entered in ASCII notation.

6-9. Determine what should be punched on a card to enter the following numbers directly in zoned-format notation:

- a. +2076
- b. -1241
- c. +1555
- d. -4909

6-10. How would each of the numbers in the previous exercise be represented in the packed decimal format?

6-11. A notable omission from the instruction set in Table 6-4 are multiply and divide instructions. Using only the instructions in Table 6-4, prepare the following:

- a. A routine to multiply two numbers stored in memory, leaving the product in the accumulator.
- b. A routine to divide one number by another, leaving the quotient in the accumulator and the remainder in a location in memory (or vice versa).

6-12. Program the problem in Exercise 2-18 using the assembly language instructions in Table 6-4 and the approach in Exercise 6-11 to accomplish multiplication. Omit READ and WRITE operations.

6-13. Program the problem in Exercise 3-22 using the assembly language instructions in Table 6-4 and the approach in Exercise 6-11 to accomplish multiplication and division. Upon completion of the routine, the binomial coefficients should be stored as an array in memory. The multiple accumulator and index register extensions to the instruction set are necessary.

6-14. Program the problem in Exercise 2-22, using the assembly language instructions in Table 6-4 and the approach in Exercise 6-11 to accomplish multiplication and division. Omit READ and WRITE operations.

In addition to the above exercises, those students learning one of the assembly language subsets in Appendix A might attempt to program one or more of the following exercises in assembly language: 2-18, 2-22, 2-30, 2-31, 2-32, 2-34 (work entirely in cents), 2-35 (work entirely in cents), 2-39, 2-40, 3-7, 3-22, 3-31, 3-32, 3-36, 3-37, 4-1 (restricted to integer values), 4-12, 4-13 (restricted to integer values), 4-14 (restricted to integer values), 4-15 (restricted to integer values), 4-25, 4-31 (work entirely in cents), 4-35, 5-2, 5-3, 5-11, 5-12, 5-13, 5-14, 5-15, 5-16, or 5-17. In some exercises, certain modifications such as deletion of printing character data are necessary.

7

THE COMPUTER SYSTEM

Digital computers come in many different models and configurations, ranging from extremely small systems costing less than \$10,000 up to very large systems costing several millions. The small systems are usually dedicated to performing a specific task on a regular basis or on demand, whereas the larger systems are capable of supporting many different tasks simultaneously. In addition to obvious hardware differences between computing systems, there is also a difference between software, that is, the programs that direct the computing system. In fact, computer systems that are identical with respect to hardware can be made to appear radically different to the user just by using different software to direct the respective operations.

This chapter begins with a discussion of hardware-related topics as they pertain to structuring a functional computing system. Later sections then give attention to the various modes in which computers operate and the required software.

7.1. THE CENTRAL PROCESSING UNIT, MEMORY, AND I/O SUPPORT SYSTEMS

As pointed out earlier, the heart of the computing system is the central processing unit (CPU) and its memory. To be able to utilize the capabilities of this hardware efficiently, it must be able to communicate with the outside world, which requires supporting systems for input/output operations. In this section, we shall describe only the supporting systems necessary to communicate to a peripheral device such as a card reader. The description of various peripheral devices is left for the next section.

Actually, the CPU is invariably equipped with an operator's console that, in addition to start and stop switches, permits the operator to examine the contents of specific locations in memory, to examine the instruction to be

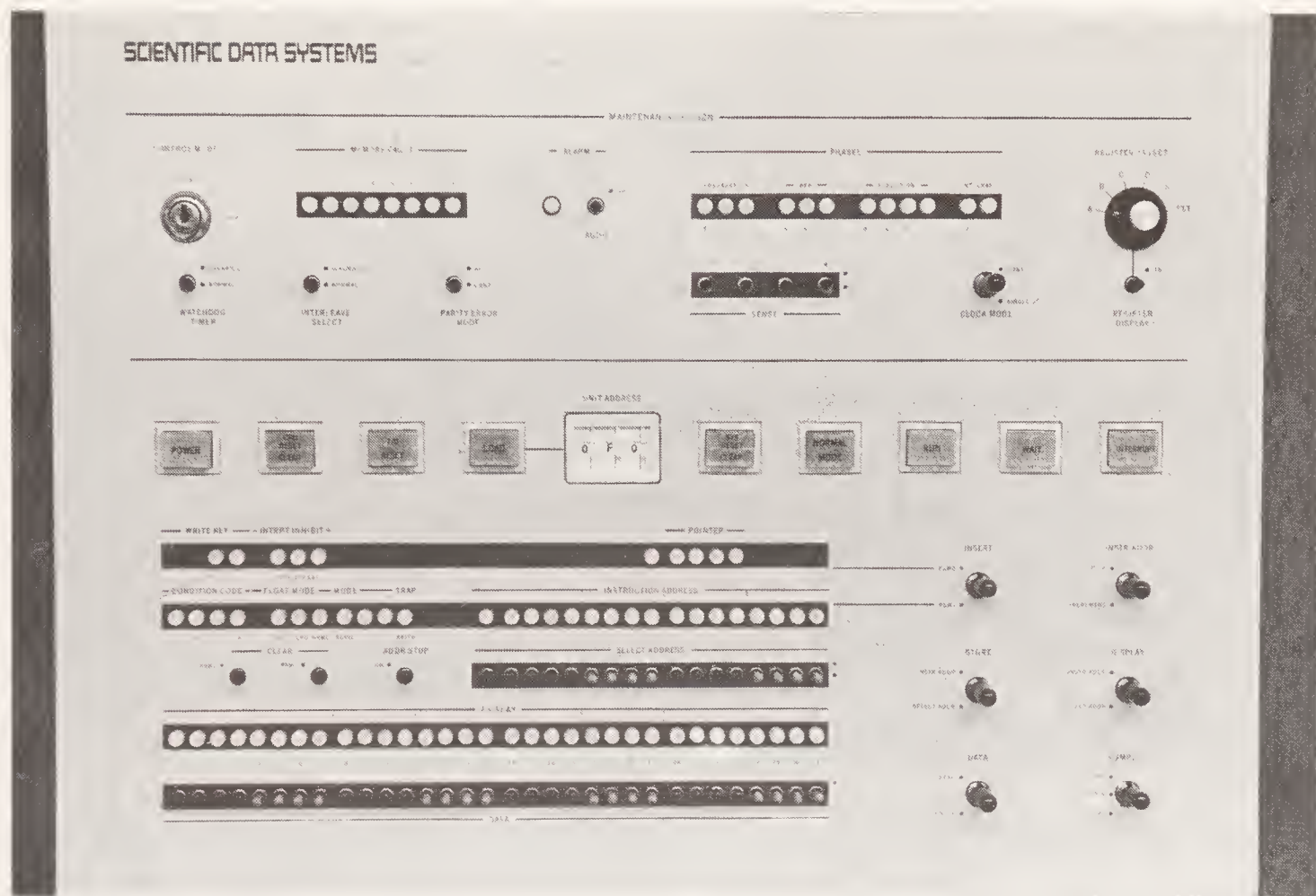


Figure 7-1 Programmer's console of the Xerox Data Systems Sigma 5 digital computer (Photo courtesy Louisiana State University)

executed, to insert information into a specific storage location, etc. Figure 7-1 shows the operator's console of a typical digital computer. The upper row of switches and lights is for the address; the bottom row of switches and lights is for the data. For example, to display the contents of a specific address in memory, the address is entered in the address switches, the "compute" switch is set to "idle," the "display" switch is set to "select address," and the contents of the address appears in the bottom row of lights.

Communication through the operator's console is obviously slow and tedious, and its use is generally limited to systems programmers trying to analyze problems with the software or to maintenance personnel trying to isolate hardware problems.

While in earlier computers the CPU was responsible for input/output operations, the direct supervision of peripheral devices in current machines is generally relegated to a special unit which we shall call the *I/O processor*. In the configuration illustrated in Figure 7-2, the I/O processor and CPU share a common access or *port* to memory. The central processing unit specifies what data are to be transmitted to or from a given peripheral unit, but the mechanics of the transfer is the responsibility of the I/O processor. The I/O processor does not interfere with the operation of the CPU except when it is transferring information to or from memory.

As illustrated in Figure 7-2, the peripheral devices are connected to the I/O processor via a common line or cable known as the *I/O bus*. This approach,

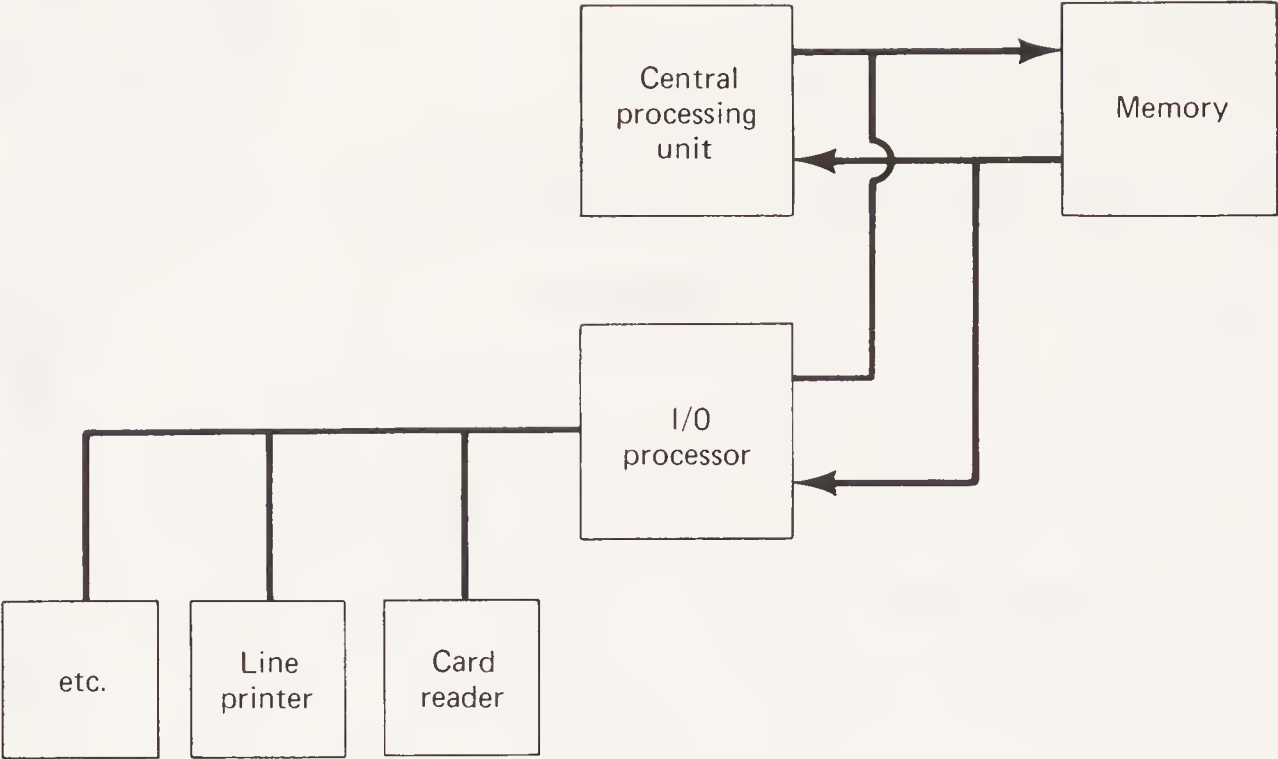


Figure 7-2 Role of the I/O processor

known as the *party-line approach*, permits the attachment of virtually any peripheral device to the I/O processor up to a specified maximum. Data transmission from the I/O processor to a peripheral device is preceded by a unique bit pattern which activates the specific peripheral unit that is to receive the desired information. All other peripheral devices ignore the data that follows. Similarly, to receive data from a peripheral device, the I/O processor sends a bit pattern that activates the desired unit, which then inserts the desired information onto the I/O bus. To keep the transmission rate as high as possible, the I/O bus consists of one wire for each bit in either a byte or a word, which permits parallel or simultaneous transmission of all bits in either a byte or a word.

The speed of the central processing unit and the I/O processor is such that several peripheral devices can be simultaneously operated on the same I/O bus. In fact, many large systems have more than one line printer and more than one card reader attached to the same I/O bus.

Furthermore, the I/O processor acts independently of the CPU in that the CPU can be performing other tasks while the I/O processor attends to input/output functions. Therefore, the CPU need not sit idle waiting for a peripheral device to function.

Some peripheral devices such as magnetic tapes are capable of extremely high data-transfer rates. Also, information is generally transferred in blocks from these devices to a specified area of memory, or vice versa. If these devices are attached to the I/O processor in the configuration shown in Figure 7-2, the central processing unit would not be able to access memory during the period that the I/O processor was transferring data at these high rates.

To alleviate this situation, many systems attach such high-speed devices to a special unit, often called a *direct memory access channel*, that can transmit data from the peripheral device directly to memory, as illustrated in Figure 7-3. The CPU dictates the location and amount of data to be transmitted from the

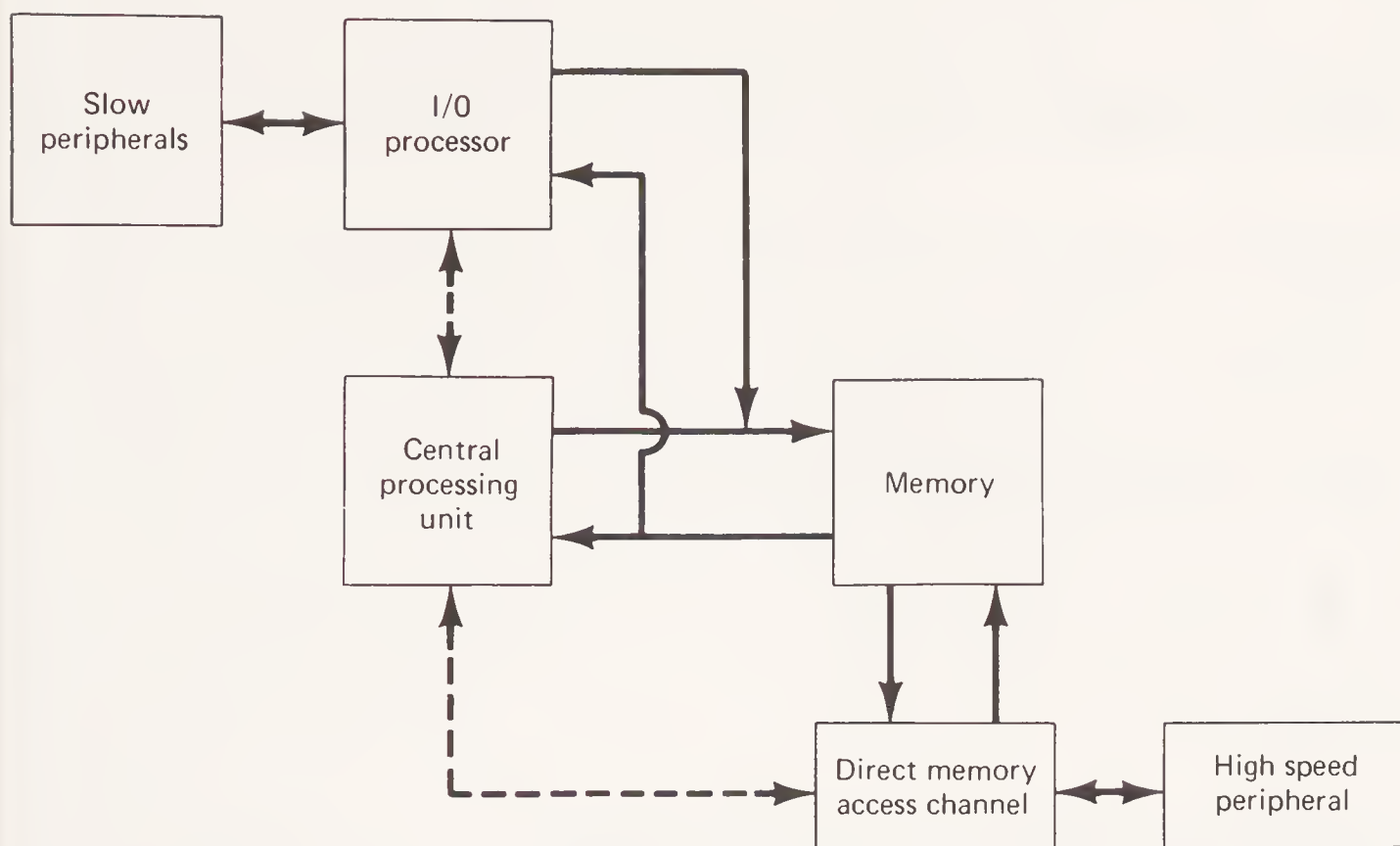


Figure 7-3 Direct memory access channel

peripheral devices, along with the location in memory into which the information is to be placed. The CPU can then attend to other tasks while the data transfer is in progress. To some extent, the direct memory access channel is nothing more than a CPU with very limited capabilities. It can transfer information, perform simple logic and test operations, and whatever other functions are necessary to do its job.

To support a direct memory access channel, the memory unit must be capable of sending data to and receiving data from two or more units. This feature is generally referred to as *multiple ports to memory*. In many systems, the I/O processor to which the slow devices are attached is also given a separate port to memory, which is the configuration presented in earlier illustrations in this book.

Figure 7-4 gives a typical I/O structure for an IBM/360 computing system. In IBM terminology, what we have called an I/O processor and direct memory access channel are referred to as a *multiplexer channel* and a *selector channel*, respectively. This configuration shows one selector channel for the tape units and another selector channel for the disk units, which are also capable of high data rates. All other peripherals are connected to the slower multiplexer channel.

In communicating with peripherals, virtually all systems use *parity* as a means of detecting errors involved in transmitting. For example, suppose the system is transmitting an eight-bit byte to a peripheral. In addition to the eight bits within a byte, the system also transmits a ninth bit known as the *parity bit*. The status of the parity bit is set so that the total number of “on” bits in the nine (including parity) bits transmitted is either even or odd. For example, suppose the system wants to transfer the character 9 in EBCDIC to a peripheral device. If the system uses even parity, the transmission would be as follows:

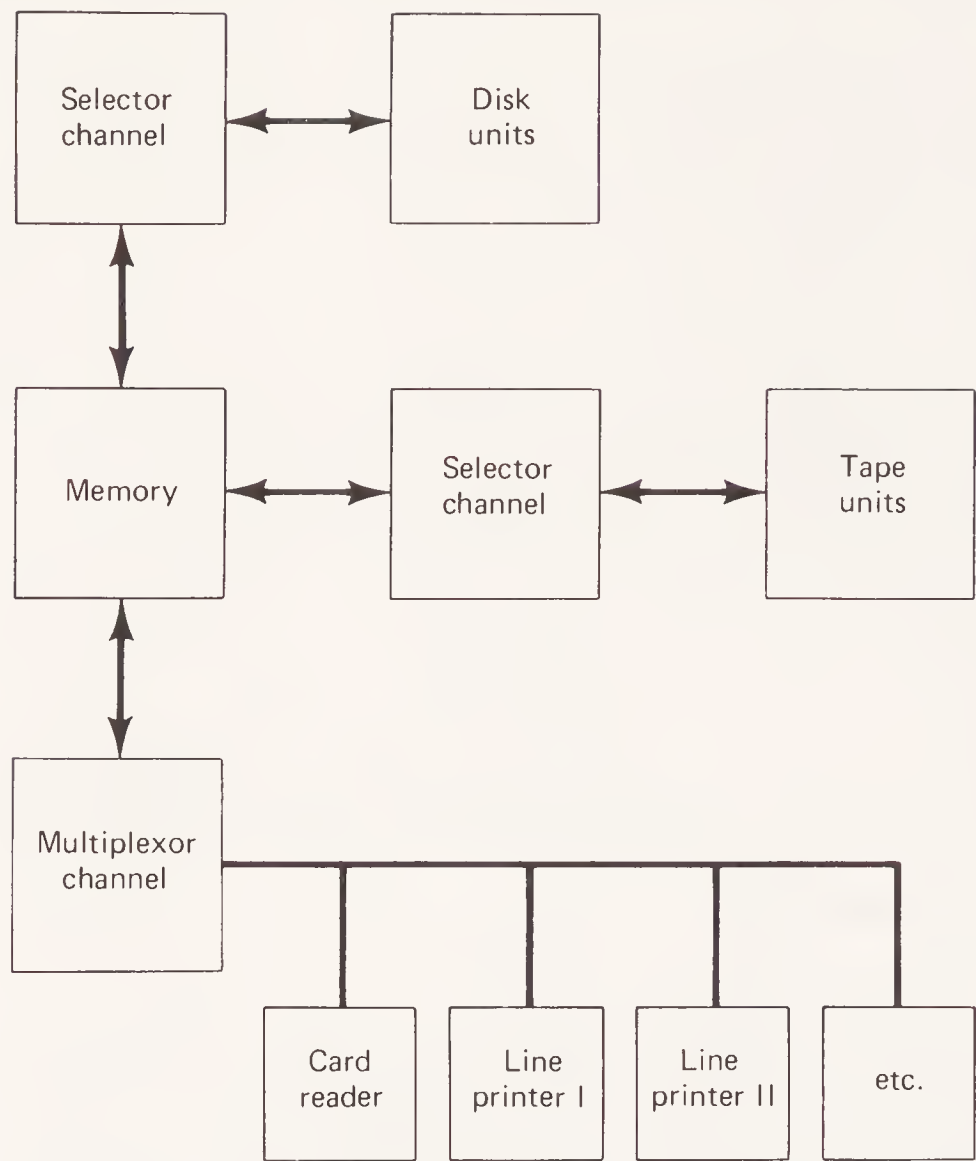


Figure 7-4 Typical I/O structure of an IBM/360 system

0 1111 1001
Parity character

For odd parity, the transmission would be

1 1111 1001
Parity character

The addition of the parity bit permits the unit receiving the transmission to verify its validity. If any one bit is changed, the unit will detect a parity check violation, which will usually result in a request for a retransmission. A parity violation does not occur if two bits are changed during the transmission, but this event is extremely unlikely.

Many systems also provide a parity bit with each word in memory. Some manufacturers, however, claim their memories are so reliable that memory parity is unnecessary.

7.2. PROCESSOR FEATURES

Just as there are differences between one automobile engine and the next, there are differences between CPUs in computers. In the last chapter, we pointed out

that different processors had different instruction sets, which affects the effective speed of the processor. On many computers, optional features can be added to extend the power of the CPU, much as accessories can be added onto a basic automobile.

Virtually all processing units are capable of directly adding and subtracting numbers in fixed-point format. Other mathematical operations, for example, multiplication of two integer numbers, can be accomplished either (1) by a software routine that uses successive additions, subtractions, shifts, etc., to accomplish the desired operation or (2) by hardware designed specifically for performing the desired operation. Use of hardware has several significant advantages, notably speed. For example, a software routine to multiply two integer numbers requires about 200 μsec as compared to about 4 μsec when hardware is used. In addition, when hardware is used, the corresponding software routine does not have to be stored in memory. The disadvantage is the added cost of the hardware.

While hardware can be designed to perform virtually any operation, it is economically feasible to include in the central processing unit only the hardware to perform frequently used operations. The three features generally considered are fixed-point hardware, floating-point hardware, and decimal arithmetic.

The term *fixed-point hardware* generally encompasses the hardware necessary to perform integer multiplications and divisions. This feature is optional only on small computers, being standard on most medium and large machines.

The term *floating-point hardware* generally encompasses the hardware necessary to perform floating-point additions, subtractions, multiplications, divisions, and other operations involving floating-point numbers. On some machines this hardware is also capable of "extended precision" floating-point operations, where to provide greater accuracy two words are used to store numbers instead of one. This feature is standard on only the very large computers, optional on most, and unavailable on some smaller machines. It is extremely desirable on machines used extensively for scientific computation.

Hardware generally referred to as *decimal arithmetic* is also available to perform arithmetic operations involving decimal numbers. Such operations as addition, subtraction, multiplication, and division of two decimal numbers are generally implemented, and some units also permit operations such as multiplication of a decimal number by an integer or fixed-point number. This feature is very desirable for machines on which substantial business data processing is performed. This feature is rarely included in the basic CPU price, and is available as an option only on machines designed for business applications.

As we have seen in earlier chapters, mathematical functions such as the square root can be implemented as a sequence of more elementary operations such as addition, subtraction, multiplication, and division. While special hardware could be constructed to compute the square root, this has not been the common approach. However, a recent development known as *read-only memory* is changing this somewhat. As the name implies, read-only memory can only be read and not written into. Therefore, most routines that can be programmed

into the conventional read/write memory can be implemented in read-only memory. In addition to the permanency of the program implemented in read-only memory, the cycle time of read-only memory is generally faster than that of read/write memory by a factor of five or more. Therefore, commonly used routines can be programmed into read-only memory to take advantage of this speed.

Another feature that has become virtually standard on recent machines is the *program interrupt*. In effect, this feature is frequently used to coordinate the operation of the computer with that of the real world. In this situation, the timing and sequence of program execution are significantly more important than in the conventional off-line data-processing machine.

As an example of this mode of operation, consider the use of computers to analyze electrocardiograms for potential heart defects. One approach is to first record the electrocardiogram on magnetic tape, then digitize the signals, and finally submit these data to the computer for analysis. In this environment, the computer can process the data at whatever time it finds convenient, as long as the results are available within a reasonable time, which may range from a couple of hours to overnight.

A more attractive approach is to use the computer to take the data directly. In this case the sensors (actually voltage detectors) are interfaced to the computer via an analog-to-digital converter to digitize the signals. The computer is then able to accept the data directly. However, the speed at which the data are generated is specified by the heartbeat rate of the patient. In the off-line case discussed in the previous paragraph, once the data were digitized, the computer could read it as fast as the peripheral device could operate. But in the on-line case, the computer must read the data at a rate commensurate with the rate at which the data is generated. Thus, the computer's operation must be coordinated with that of the outside or real world. Computers functioning in this manner are said to be operating in *real time*.

The feature generally used to achieve this coordination is the interrupt. In response to an interrupt signal, the computer can immediately suspend the execution of a program in progress, execute a particular block of code (called an *interrupt service routine*) associated with the particular interrupt received, and then resume execution of the suspended program. The mechanics of this operation are as follows:

1. An interrupt signal is received.
2. The contents of the program counter (current address register) are stored in a preassigned location.
3. The program counter is loaded with the address of the first instruction in the interrupt service routine.
4. The first few instructions in the interrupt service routine must store any registers that will be used in executing the interrupt service routine.
5. The service routine is executed.

6. The last instructions in the interrupt service routine must restore the registers to their original values.
7. The program counter is restored to its value at the time the interrupt occurred.

While this process is basically simple, a great deal of flexibility is available, especially on machines that permit some interrupts to “interrupt” the service routines of other interrupts.

7.3. PERIPHERAL UNITS

In this section we shall briefly describe the various peripheral input/output units commonly found on modern digital computers.

Typewriter or Teletype

A relatively inexpensive unit for low-volume input or output is the *typewriter* or *teletype*. (See Figure 7-5.) These units have characteristically been mechanical in



Figure 7-5 Keyboard terminal with optional low-speed paper-tape unit
(Courtesy Univac Corporation)

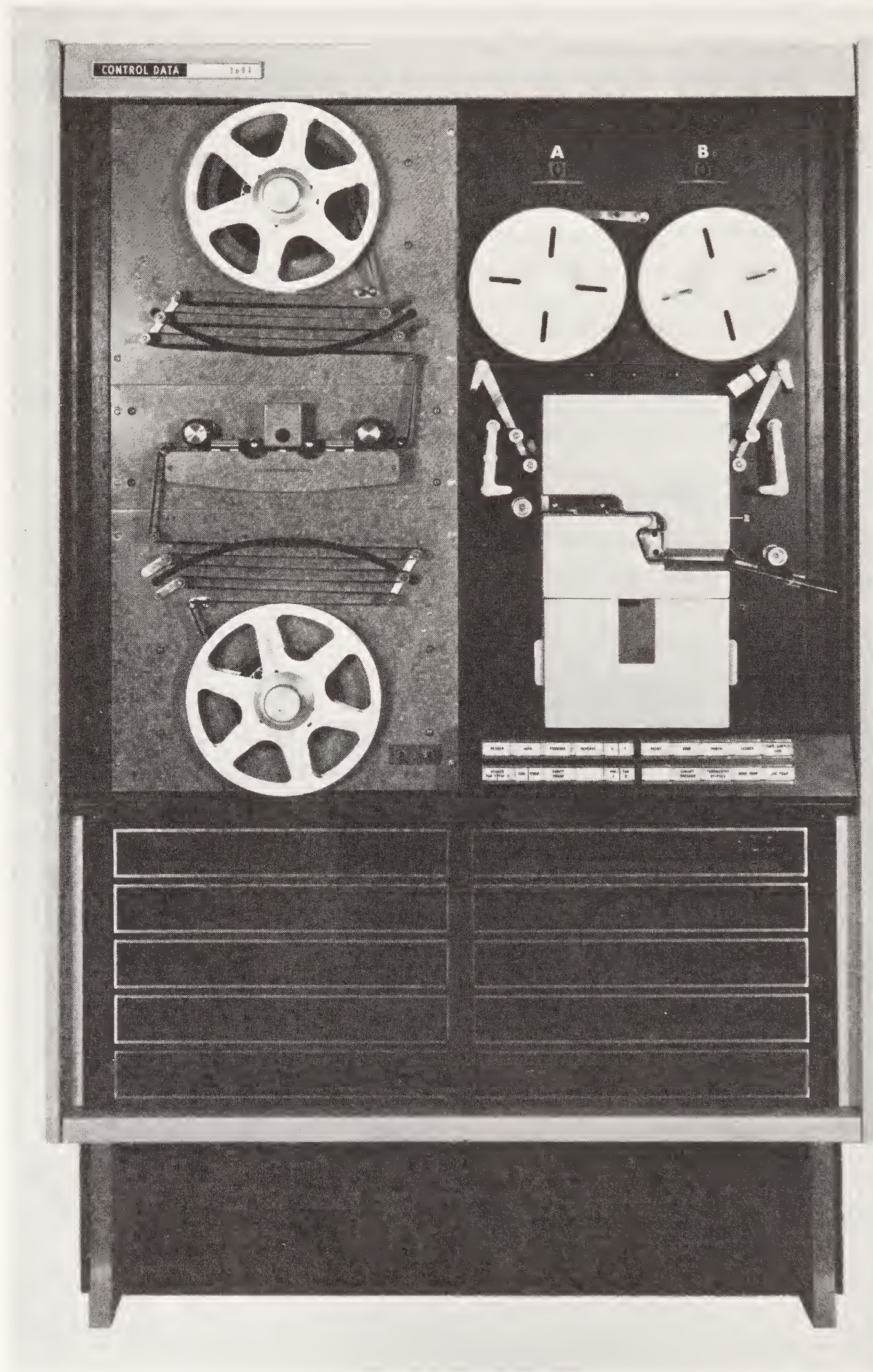


Figure 7-6 High-speed paper-tape unit (Courtesy Control Data Corporation)

nature and operated at about 10 characters per second. Recent developments, however, have led to other approaches for printing, for example, using a dot matrix to form the characters, and the speeds have increased to about 30 characters per second (and the associated noise reduced considerably).

Teletypes and typewriters are frequently used as consoles via which computer operators communicate with the processor. They are also very popular for use as individual terminals in time-sharing networks.

Paper Tape

As illustrated by the teletype in Figure 7-5, a slow-speed paper-tape unit can be added at a relatively modest cost. Although this device can read and punch paper tape, its speed is again on the order of 10 characters per second. For large volumes of input or output via paper tape, high-speed units are available (Figure 7-6) that operate at about 300 characters per second reading and 100 characters per second punching. Figure 7-7 illustrates a typical character code on punched paper tape.

Line Printer

Whereas the typewriter or teletype prints one character at a time, the line printers (Figure 7-8) print one line at a time and thereby are more attractive when the volume of output is high. The number of characters per line varies depending upon the model, but usually is about 132 characters. The printing speed is measured in lines per minute. The slower printers used on small computers operate at speeds starting as low as 80 lines per minute. Larger computers have line printers with speeds between 600 and 1,100 lines per minute, usually closer to the latter figure. These are generally mechanical devices that print on ordinary paper with sprocket holes along the edges so that it can be guided through the printer. Much higher speeds can be obtained by devices using special paper that is sensitive to heat or light, but the additional cost of the special paper has apparently made this approach uneconomical.

Card Read/Punch

To date the most popular medium in which to prepare programs has been punched cards. (See Figure 7-9). Therefore, most computers of any size have a card reader and a card punch. The speed of the card readers range from as low as 100 cards per minute up to 1,000 cards per minute or more. Card punches are slower, the faster ones currently available operating at speeds of about 400 cards per minute.

Magnetic Tape

While punched cards provide a convenient medium for many purposes, they have some limitations. While they are not especially expensive, nevertheless they are not reusable (that is, once they have been punched with particular data, they

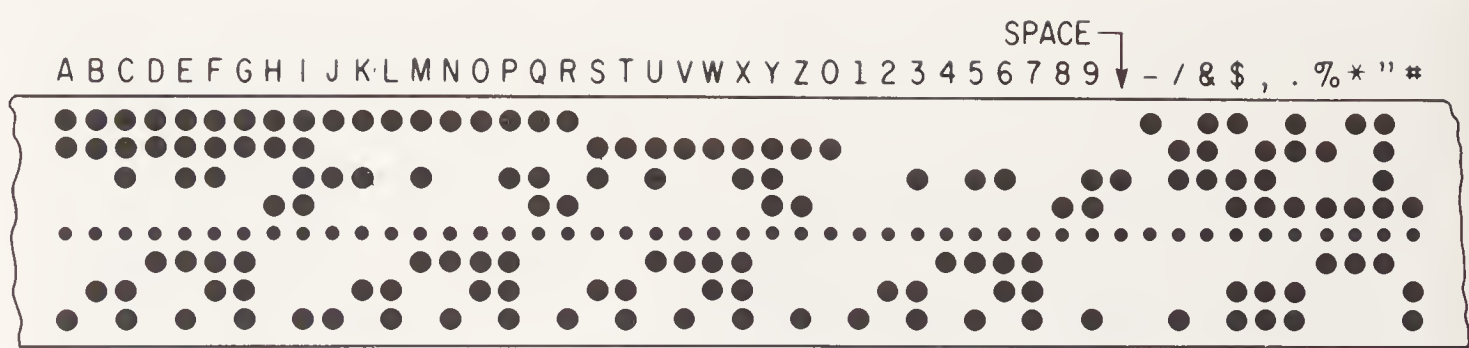


Figure 7-7 Paper-tape character code

cannot be repunched with other data). In addition, storage of large amounts of data in the form of punched cards is quite space-consuming. Magnetic tape provides a much more concise medium for storage of large amounts of data, as well as being reusable (that is, magnetic tape can be erased and written upon again). In addition, the input and output rates are substantially higher than for cards.

In reality, the magnetic-tape units used on computers are little more than a digital version of the home tape recorder. (See Figure 7-10.) The tape is generally 1/2 inch in width and consists of a plastic base coated with a magnetic oxide. The width of the tape is divided into either seven or nine subdivisions called *tracks*. Machines using a six-bit character code generally have seven-track tapes, which enable the six bits in the character plus a parity bit to be written in parallel on the tape. Machines using an eight-bit character code generally have nine-track tapes. A character code on magnetic tape is illustrated in Figure 7-11.

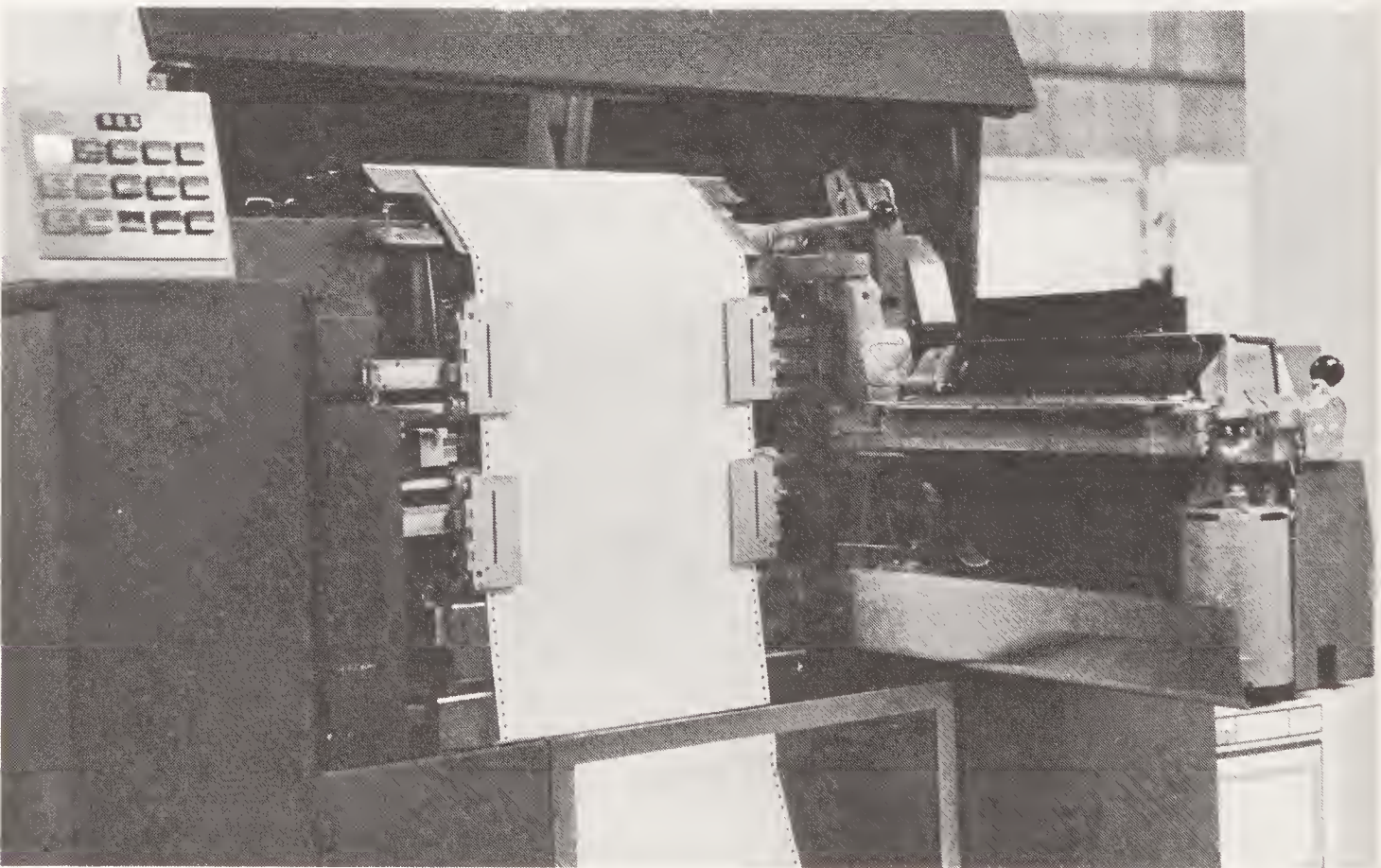
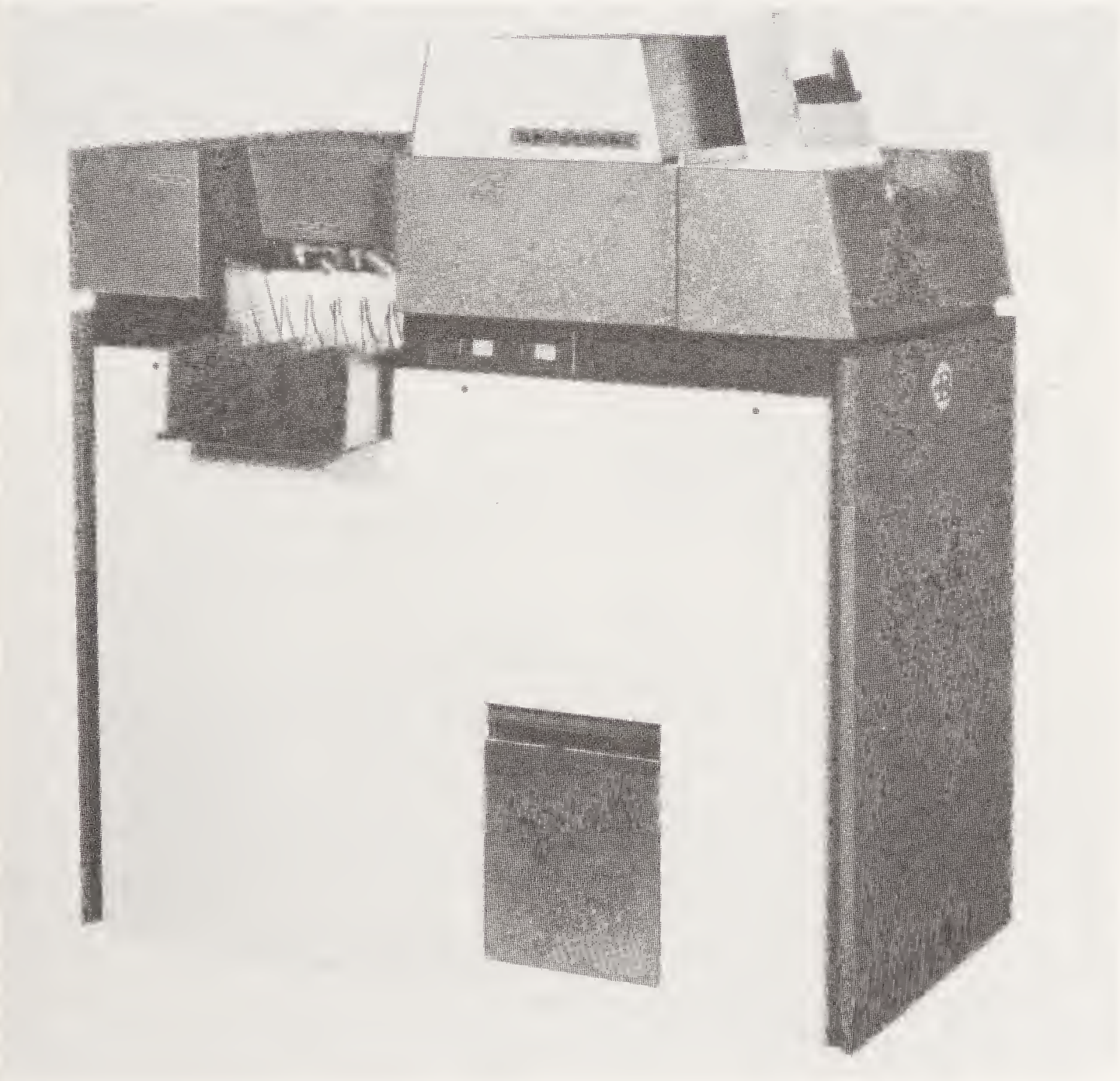


Figure 7-8 High-speed line printer (Photo courtesy Louisiana State University)



a. Card reader, 1100 cds/min (Courtesy Louisiana State University)



b. Card punch, 300 cds/min (Courtesy Burroughs Corporation)

Figure 7-9 Card read and card punch unit



Figure 7-10 Magnetic-tape units (Courtesy Louisiana State University)

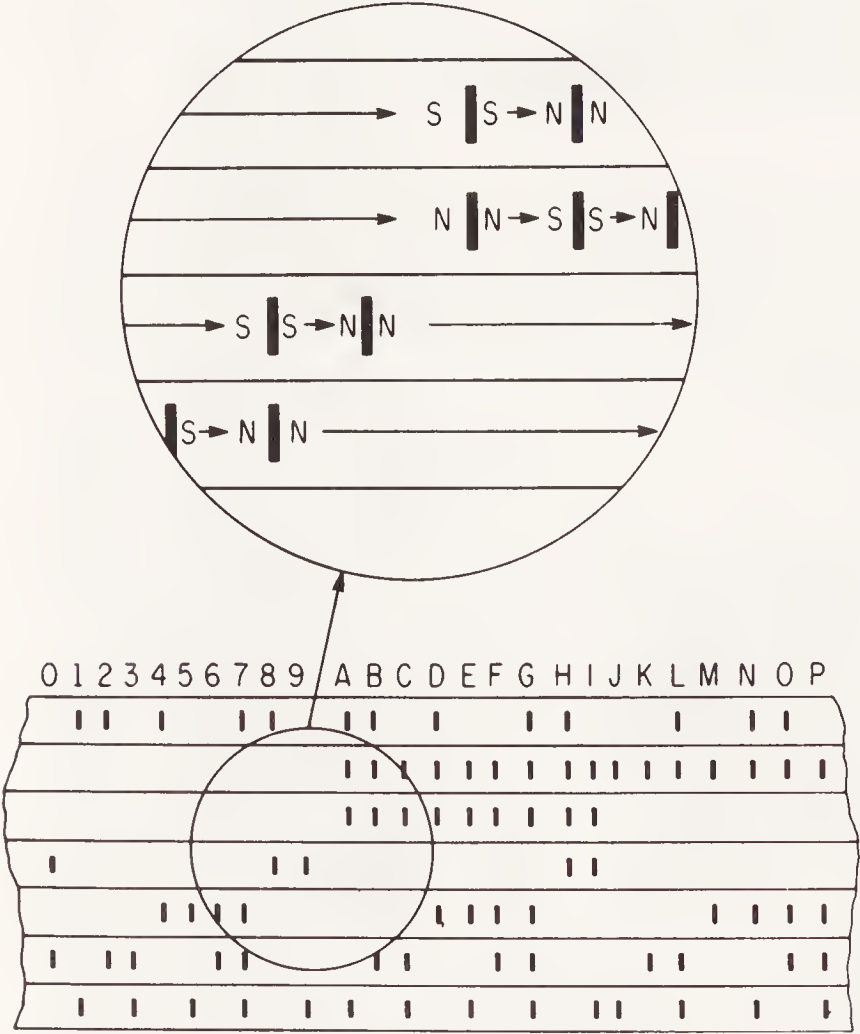


Figure 7-11 Character code on magnetic tape

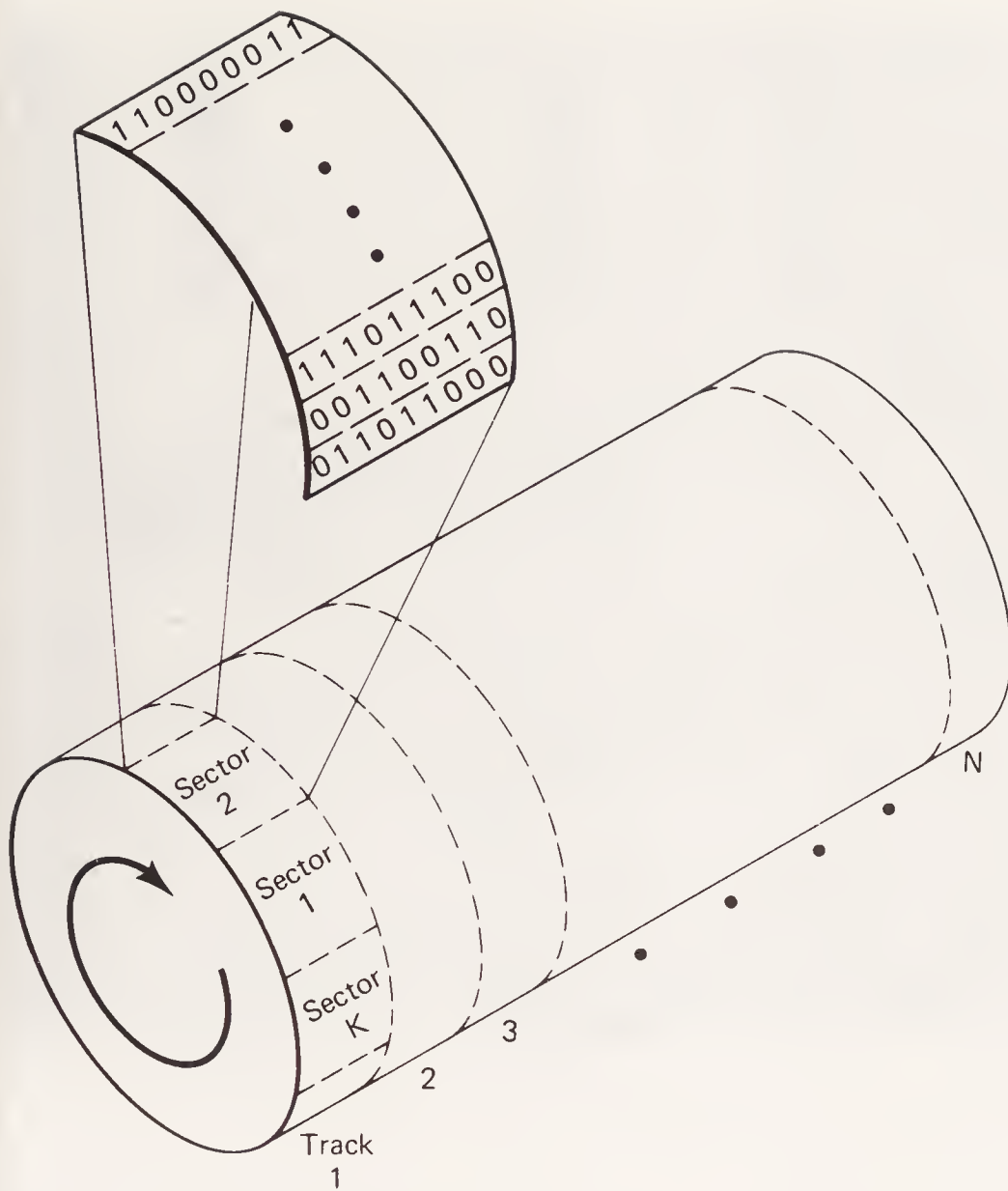


Figure 7-12 Drum layout

Reels up to 2,400 feet in length may be used with writing densities up to 2,400 bpi (bits per inch). Consequently, a single magnetic tape may store up to 45 million characters. Data-transfer rates are also high, with rates up to 320,000 bytes/sec being common.

Drum

While the magnetic tape can store large quantities of data which can be read or written very rapidly, it must be in a sequential manner. That is, if the read/write head is currently positioned near the beginning of the tape and the data to be read is near the end of the tape, the tape must be read in a sequential manner until the desired data is reached. The time required may be as much as 2 or 3 minutes.

These disadvantages are circumvented by use of a drum. As illustrated in Figure 7-12, the drum basically consists of a cylinder coated on the outside with magnetic material. The drum rotates quite rapidly, making one revolution in 33 milliseconds. The drum is divided into tracks, each of which is provided with an individual read/write head. The tracks are further subdivided into sec-

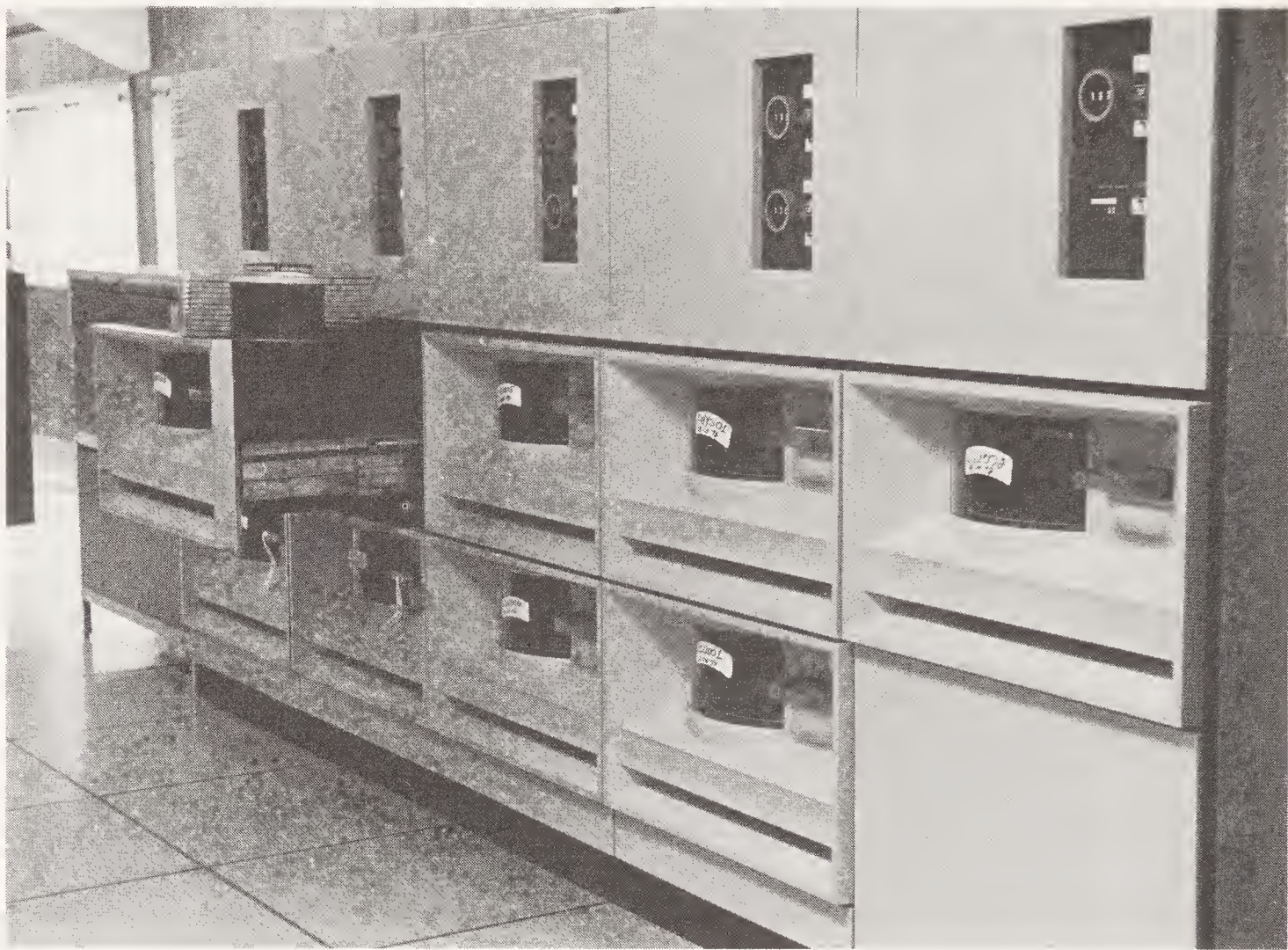


Figure 7-13 Disk unit with removal disk cartridge above extended drawer (Photo courtesy Louisiana State University)

tors. Therefore, any item of data is no more than 33 milliseconds from a read head, which makes the access time for the drum substantially less than for the magnetic tape. A typical drum may store about 3 million characters, which may be read or written at high rates. But unlike the magnetic tape, the drum cannot be readily dismounted and replaced by another one.

Disk

To some extent, the disk (Figure 7-13) is a compromise between the magnetic tape and the drum. The access time is somewhat slower than for the drum, but the magnetic surfaces may be readily interchanged.

The disk essentially consists of one or more magnetically coated flat plates which rotate at high speeds (about 33 milliseconds per revolution). Like the drum, each plate is subdivided into tracks (Figure 7-14), which are further divided into sectors. But instead of one read/write head per track, a single head is provided with a positioning mechanism to move the head from track to track as necessary. Thus, the access time for the disk is essentially the access time for the drum plus the time required to reposition the read head. However, the read head may be completely retracted to permit the disk cartridge or “disk pack” to be removed and replaced by another one.

Disks can also be manufactured with one read/write head per track. Often called “drisks,” these units have essentially the same characteristics as drums. Specifically, the writing surfaces are not removable, but the access time is equivalent to that of the drum, (Figure 7-15).

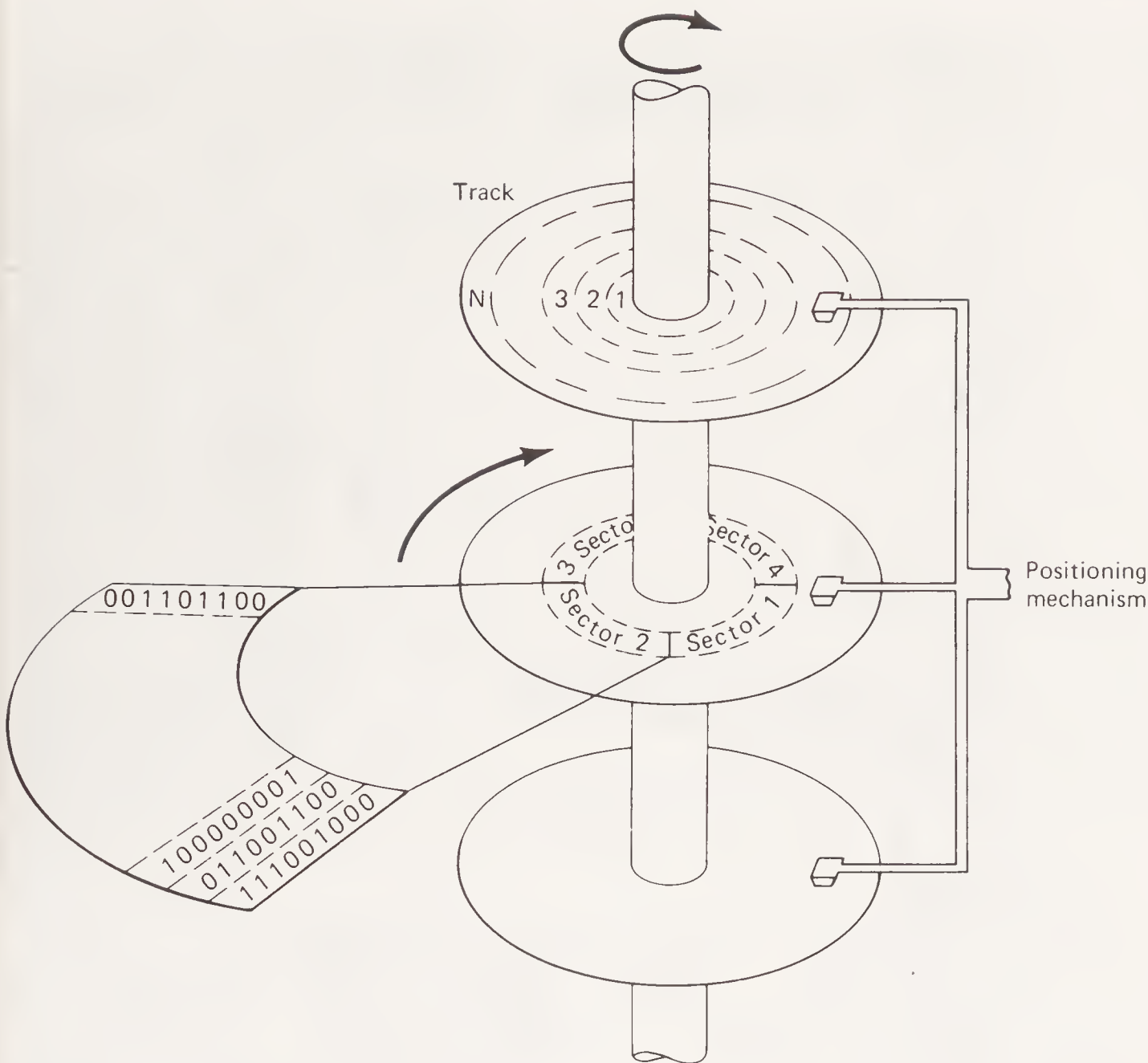


Figure 7-14 Disk layout

Incremental Plotters

In many cases, the computer output is most useful when in the form of a drawing or graph. The incremental plotter (Figure 7-16) has proven to be an extremely useful device by which such drawings or graphs can be prepared directly by the computer. Although such devices are often capable of making only the very short movements illustrated in Figure 7-17a, quite elaborate plots can be made with the proper succession of these. A relatively simple one is illustrated in Figure 7-17b.

Display Units

The use of CRT (cathode-ray tube) units to display information generated by the computer is extremely attractive for some applications. Basically, CRT devices fall into three different categories:

- 1. *Alphanumeric CRT's* (Figure 7-18a). The least expensive CRT's cost about \$2,000 (1971 prices) and are capable of displaying only characters. They are generally equipped with a keyboard which essentially makes

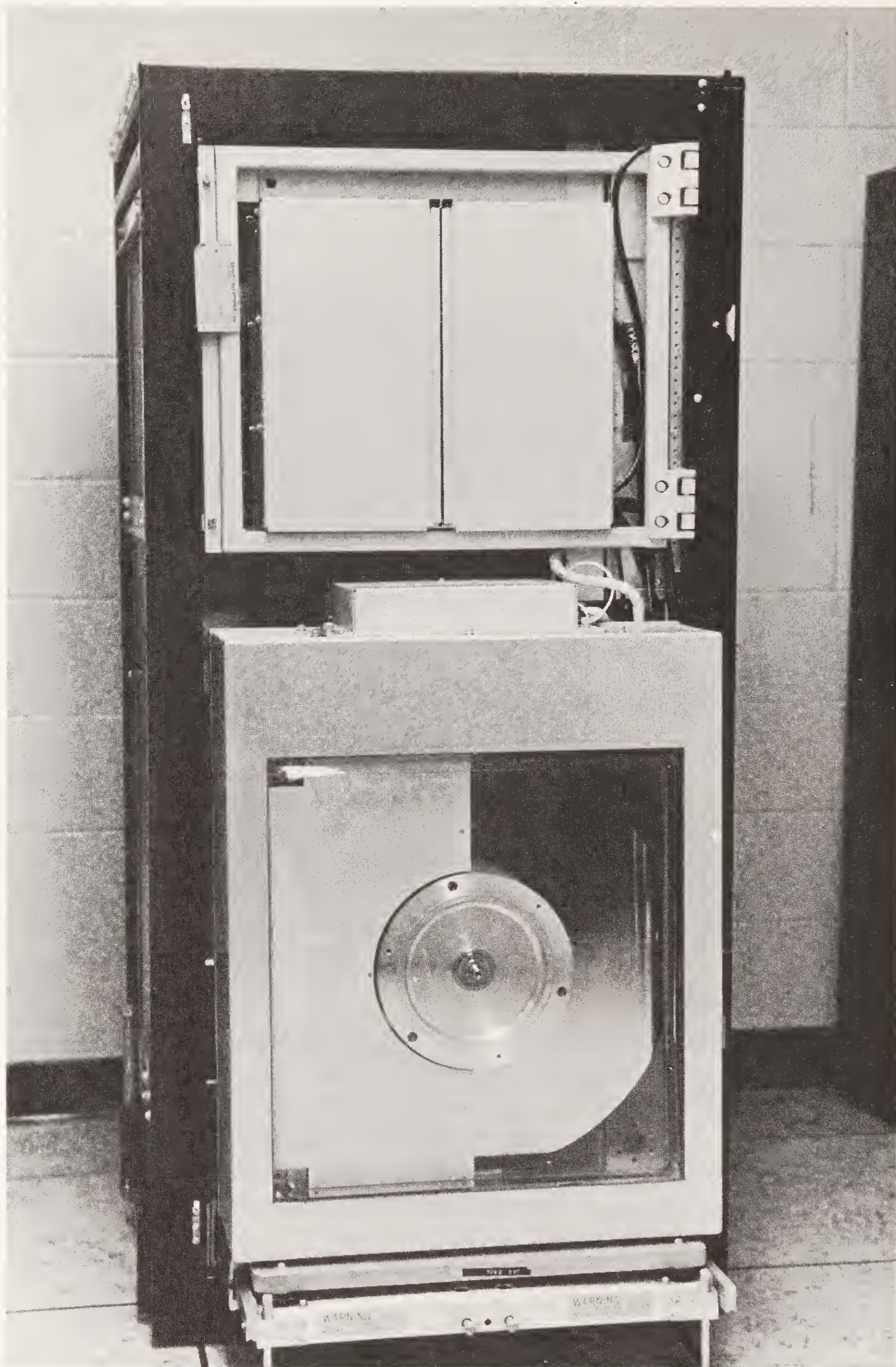


Figure 7-15 Disk with storage unit at bottom, electronics above (Photo courtesy Louisiana State University)

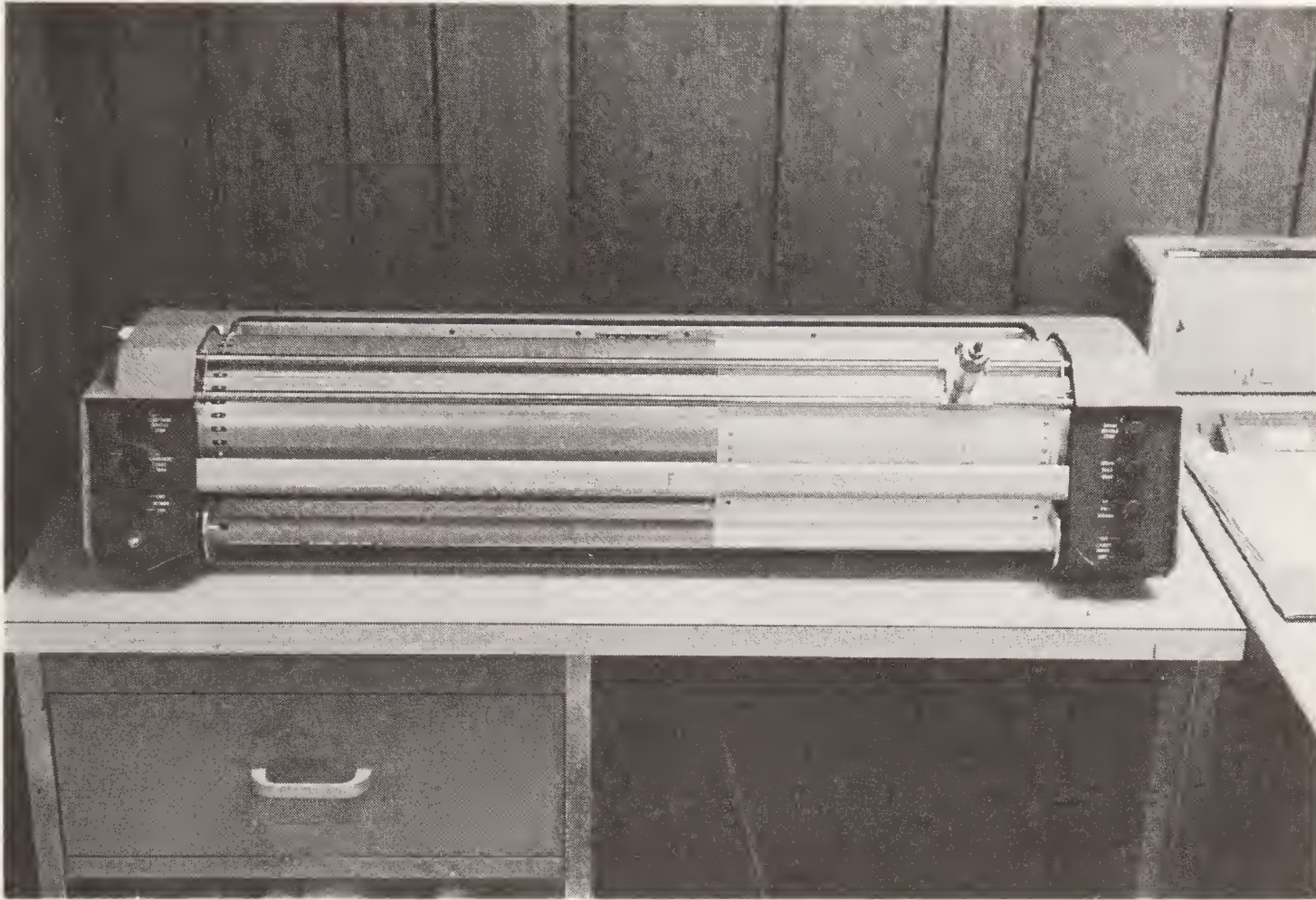
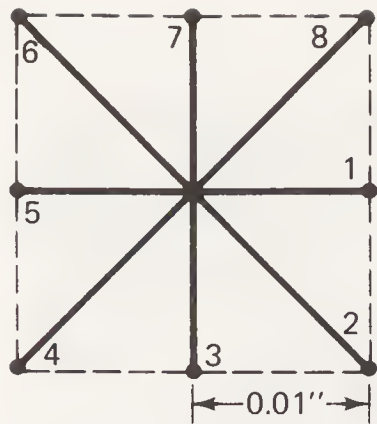


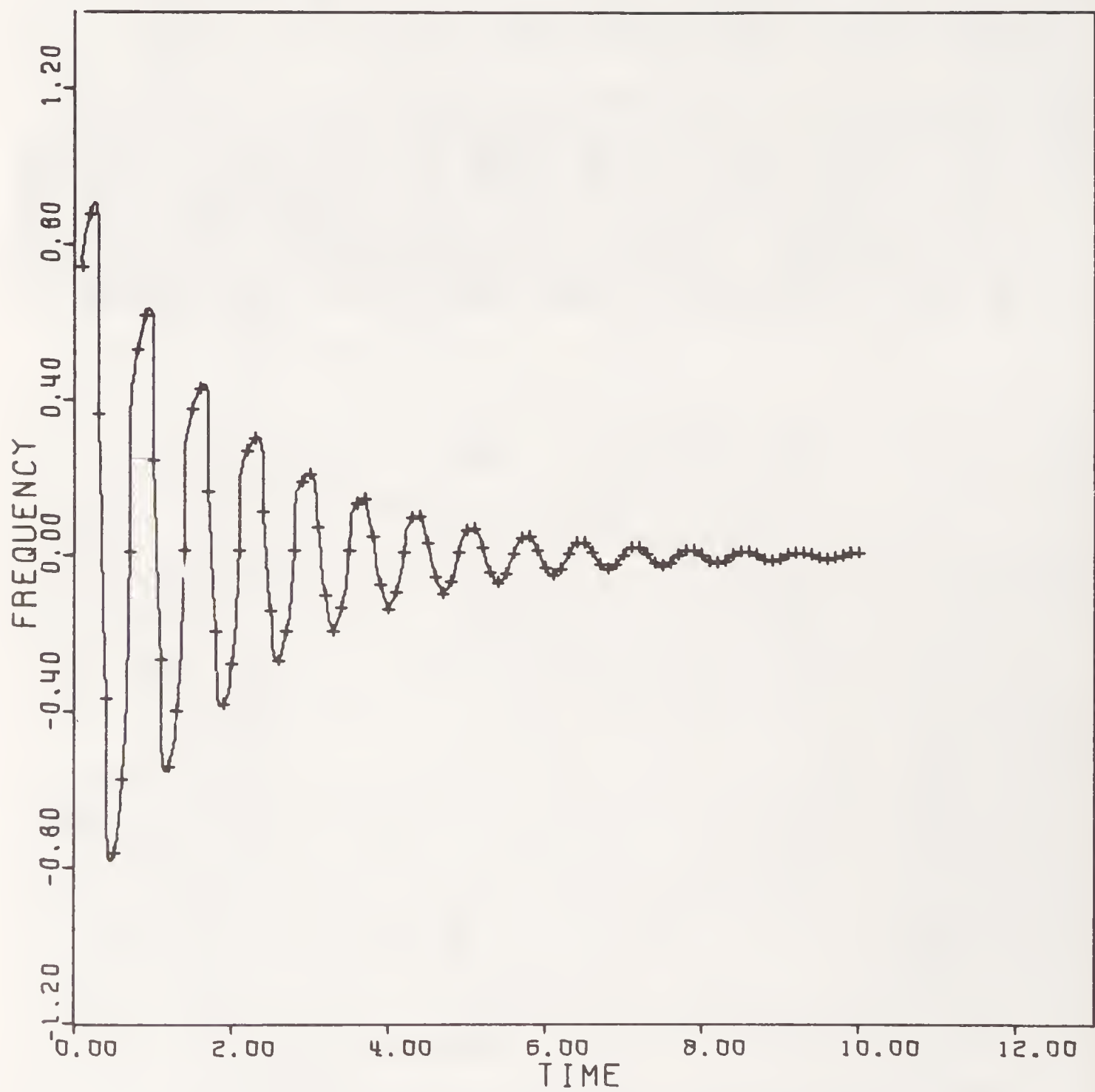
Figure 7-16 Incremental plotter (Photo courtesy Louisiana State University)

them the counterpart of the typer or teletype. They are capable of much higher data-display rates than the teletypes. The fact that they do not directly produce a “hard copy” (that is, something that can be kept for posterity) can be an advantage or disadvantage depending upon the application. While devices are available for making a copy of whatever appears on the CRT, they are relatively expensive. Many applications, for example, display of the gate and current status of airline flights in airport terminals, do not require a hard copy and are thus ideal for alphanumeric CRT’s.

2. *Storage-Tube CRT*. Many applications require the display of curves or figures, which cannot be directly accomplished with the alphanumeric CRT. The storage-tube CRT essentially retains whatever information that is “drawn” upon its screen until “erased.” Figures can be drawn by passing the electron beam over the screen in much the same fashion as the incremental plotter moves the pen over the paper. Thus, the graph in Figure 7-17b could be readily drawn on a storage-tube CRT. The entire image can remain on the screen for relatively long periods of time, often up to 30 minutes. More information can be added to the screen at any time, but in order to remove any part, the entire screen must be erased. In 1971 prices, storage-tube CRT’s cost around \$10,000.
3. *Refresh-Mode CRT’s* (Figure 7-18b). In some situations, it is desirable to display motion on the screen. For example, it is possible to display a picture of a building on the screen of a refresh-mode CRT and then



Movement 0: Pick pen up
 Movement 9: Put pen down
 a. Plotter-pen movements



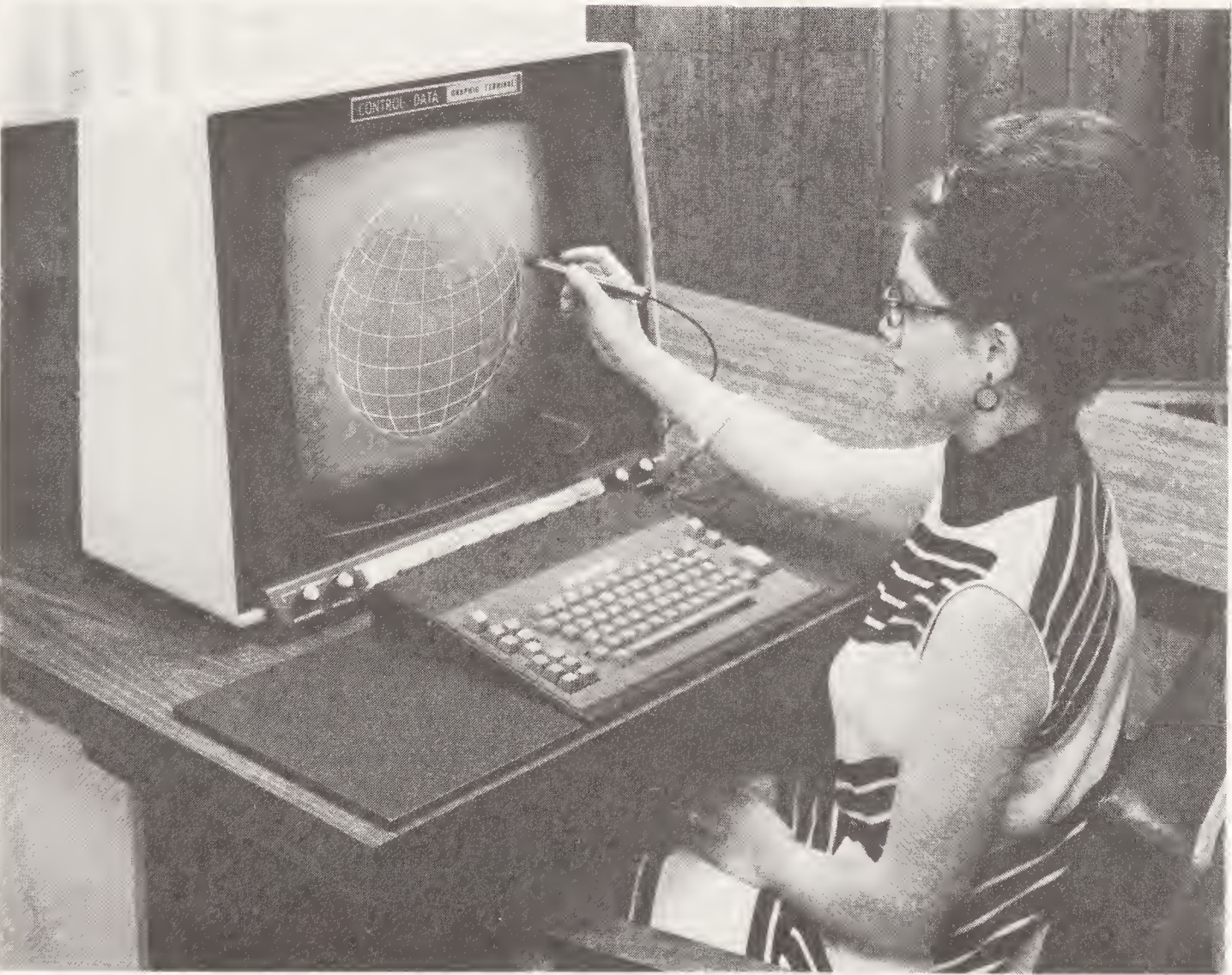
DAMPED OSCILLATING CURVE

b. Typical plot

252 Figure 7-17 Incremental plotter operation



a. Alphanumeric CRT (Courtesy Sperry Rand)



b. Refresh-mode CRT (Courtesy Control Data Corporation)

Figure 7-18 Cathode-ray-tube units

“rotate” the building so that it can be viewed from various angles. The image on the screen of the refresh-mode CRT is “refreshed” very rapidly, in much the same fashion as is done in the home television. The user of the refresh-mode CRT can often use a “light-pen” to draw upon the screen, move images around, etc. While these devices are very attractive, their price (about \$60,000) and the associated computational load on the computer have limited their use. Recent innovations, however, are reducing the price considerably, which will certainly increase their utilization.

Others

The above list of peripheral devices is not complete and could be expanded to include such devices as optical scanners and magnetic ink readers. The spectrum of peripheral devices for computers is expanding rapidly along with improvements in existing devices.

7.4. THE USE OF COMPILERS

In this chapter and the previous one, we have examined hardware features that provide the capability to accomplish a wide variety of tasks. Each individual machine-language instruction that the computer can directly execute is in itself generally a small accomplishment, but when placed in the proper sequence of instructions, the result can be significant. The problem is somehow to specify the sequence of instructions appropriate for the problem to be solved.

A further constraint is that this must be accomplished with a reasonable amount of effort. Of course, what is “reasonable” varies with the problem. Those whose solution generates a significant reward either economically or otherwise can justify a considerable expenditure of time and effort. Less time can be spent on problems whose solution is less rewarding.

While all problems could be programmed directly in machine language, this approach is extremely time-consuming. Use of assembly language generally provides an acceptable alternative but is also relatively time-consuming. Current practice is to utilize problem-oriented languages such as Fortran or Cobol, resorting to assembly language only when the programmer must have access to some basic capability of the computer that has been omitted from the problem-oriented languages.

As described in Chapter 1, the compiler translates the statements in the respective programming language into either machine language or into assembly language. In the latter case, the computer’s output is then input to the assembler, which generates the machine language instructions. Use of a compiler that directly generates machine language is illustrated in Figure 7-19. It is assumed that this computer only has a card read/punch and a printer. The first step is to clear the machine and enter the compiler via a deck of cards. The

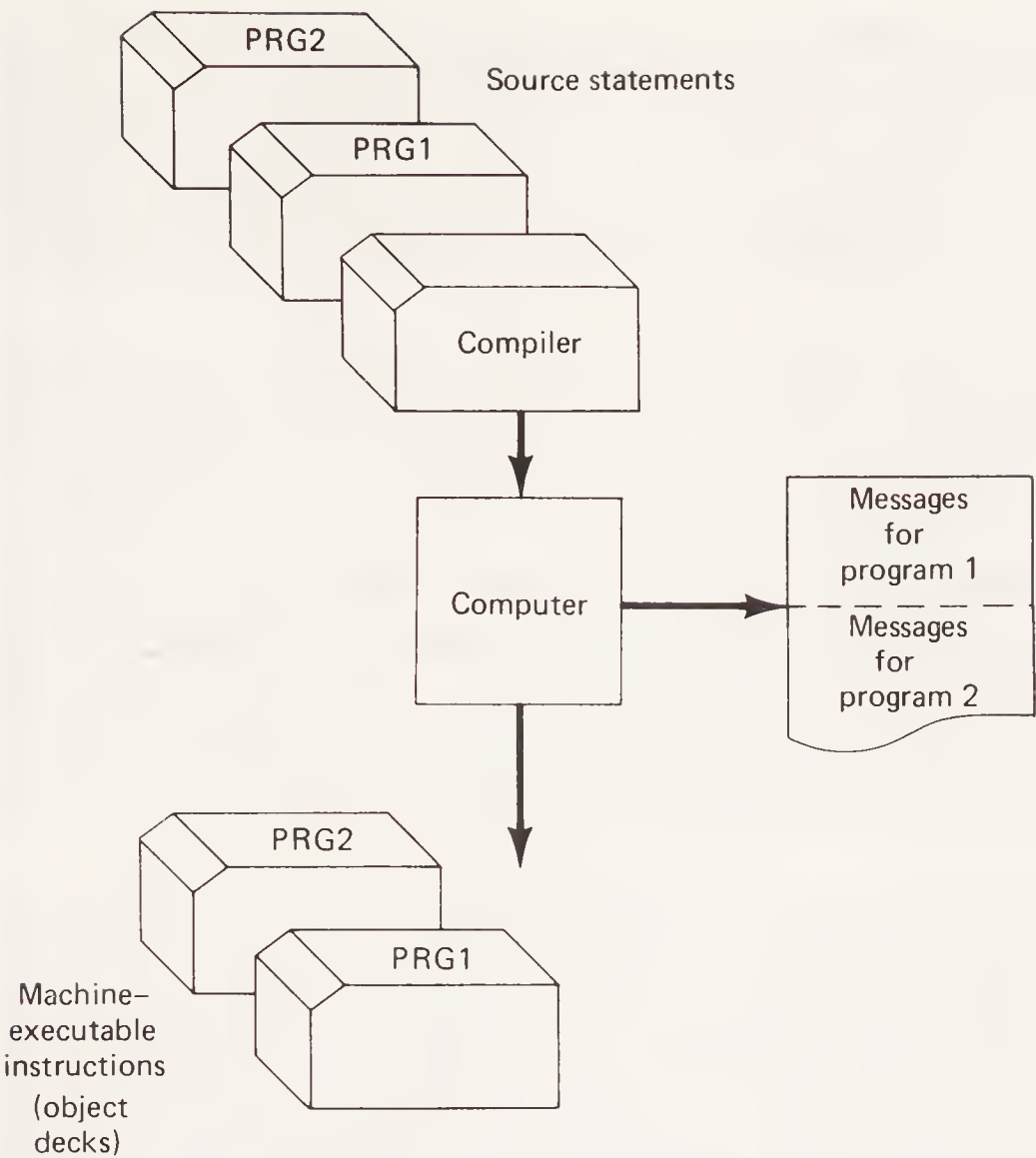


Figure 7-19 Operation of a compiler

compiler is followed by another deck of cards containing the source statements. The compiler generates messages to the operator on the printer and, unless there are significant coding errors in the source statements, punches a card deck containing the machine-executable instructions.

Since loading the compiler is not a trivial operation on computers of this type, the compilers are generally capable of compiling as many source programs as are available, which saves reloading the compiler itself. Figure 7-19 illustrates the compilation of two source programs. This feature is especially useful when the program consists of a main program and one or more associated subprograms. All can be compiled with only one loading of the compiler.

Although the output of the compiler is a set of machine-executable instructions, they cannot be executed directly because all addresses referred to in the instructions are relative to the start of the program as opposed to absolute addresses. The problem is that the compiler generally does not know exactly where the program will reside in memory. This is especially the case with subprograms.

In most systems, the lower area of memory contains system or executive routines that do such tasks as communicate with the peripheral devices. The main program generally follows immediately. Since the first position available to the main program is known, its location could be given to the compiler, which

could in turn generate directly executable machine instructions. This is beset with certain problems. While the system of executive routines is not changed very often, modifications must occasionally be made, which in turn changes the first location available to the main program. If this occurs and if the compiler generates directly executable machine instructions, all programs must be recompiled.

In the case of subprograms, the length of the main program must also be known to specify where the subprogram will be stored. If the main program were compiled and immediately followed by compilation of the subprogram, this location could be easily determined. But, further, suppose the main program and subprogram are loaded into the computer, and upon executing the programs, an error in the main program is detected. The main program must now be modified and recompiled, which will generally change the length of the main program, which in turn causes the subprogram to begin at a different point in memory. If the subprogram was compiled with absolute addresses, it must now be recompiled.

To circumvent this problem, the compiler's output contains relative addresses, which are changed to absolute addresses by a program known as a *loader*. The stream of decks associated with this phase of the process is illustrated in Figure 7-20. The first function of the loader is to place the main program and subprogram into memory, changing all relative addresses to direct addresses. The final step is to load all library routines called by either the main program or the subprogram. Generally, the complete set of library routines is passed through the reader, and the loader selects those required and places them into memory. The final result is a program ready to be executed.

As pointed out in Chapter 1, an alternative to this process is to use a

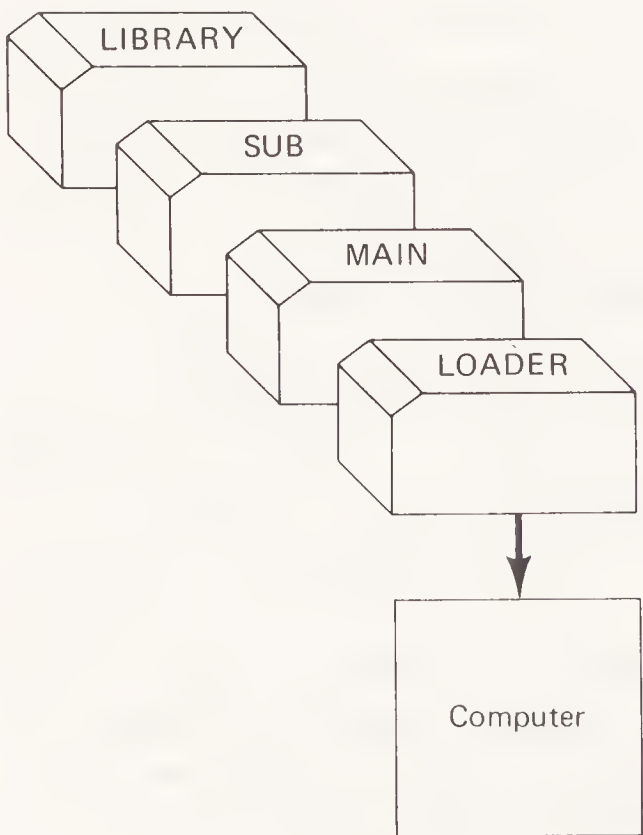


Figure 7-20 The loader phase

load-and-go type of compiler. In this type of operation, the compiler is placed in memory and remains there at all times. The instructions generated by the compiler are placed directly into memory immediately following the compiler itself. As the library routines are generally an integral part of the load-and-go type of compiler, the program is ready to be executed.

The load-and-go compiler has several advantages and disadvantages as compared with the compilation-loader type of operation:

1. Since the load-and-go compiler remains in memory at all times, less space is available for the user's program.
2. Since the instructions are not punched, the main program and all subprograms must be recompiled each time the program is run.
3. Compilation time for the load-and-go type of compiler is generally much less than for the compiler that outputs machine-executable instructions. In addition, the loader phase is completely absent. But, in turn, the load-and-go compilers generally produce less efficient code, which results in a longer program execution time.
4. The typical load-and-go compiler produces better error diagnostics, especially for errors encountered during the execution phase.

A net result of this is that the load-and-go compiler is generally highly preferred for student-type programs, where the program is short, is frequently changed, executes quickly, and good error messages are essential. On the other hand, production programs (for example, the payroll program) that are used repeatedly and changed only rarely are generally compiled by compilers that produce machine-executable instructions. This set of instructions is saved from run to run to avoid recompilation.

7.5. BATCH-PROCESSING SYSTEMS

In the last section, we discussed the various operations that must proceed in order to compile a program. While the various compilers and loaders were frequently manually loaded into computers in the late 1950s, as the speed of the computers continued to increase, this process quickly became the factor that limited the computer's throughput, that is, the number of jobs that the computer ran per hour.

In order to increase the throughput, the process of loading the various compilers and other similar material was relegated to the computer itself. In order to achieve this, two things must be added to the card-oriented computer discussed in the last section. First, a peripheral unit that can be manipulated automatically must be added. In early configurations, magnetic tapes were commonly used for this purpose. All the compilers and other supporting programs were loaded onto a magnetic tape. If, for example, the Cobol compiler was needed, the computer searched the tape (from the beginning) until the Cobol compiler was located, loaded the compiler, and then rewound the tape so

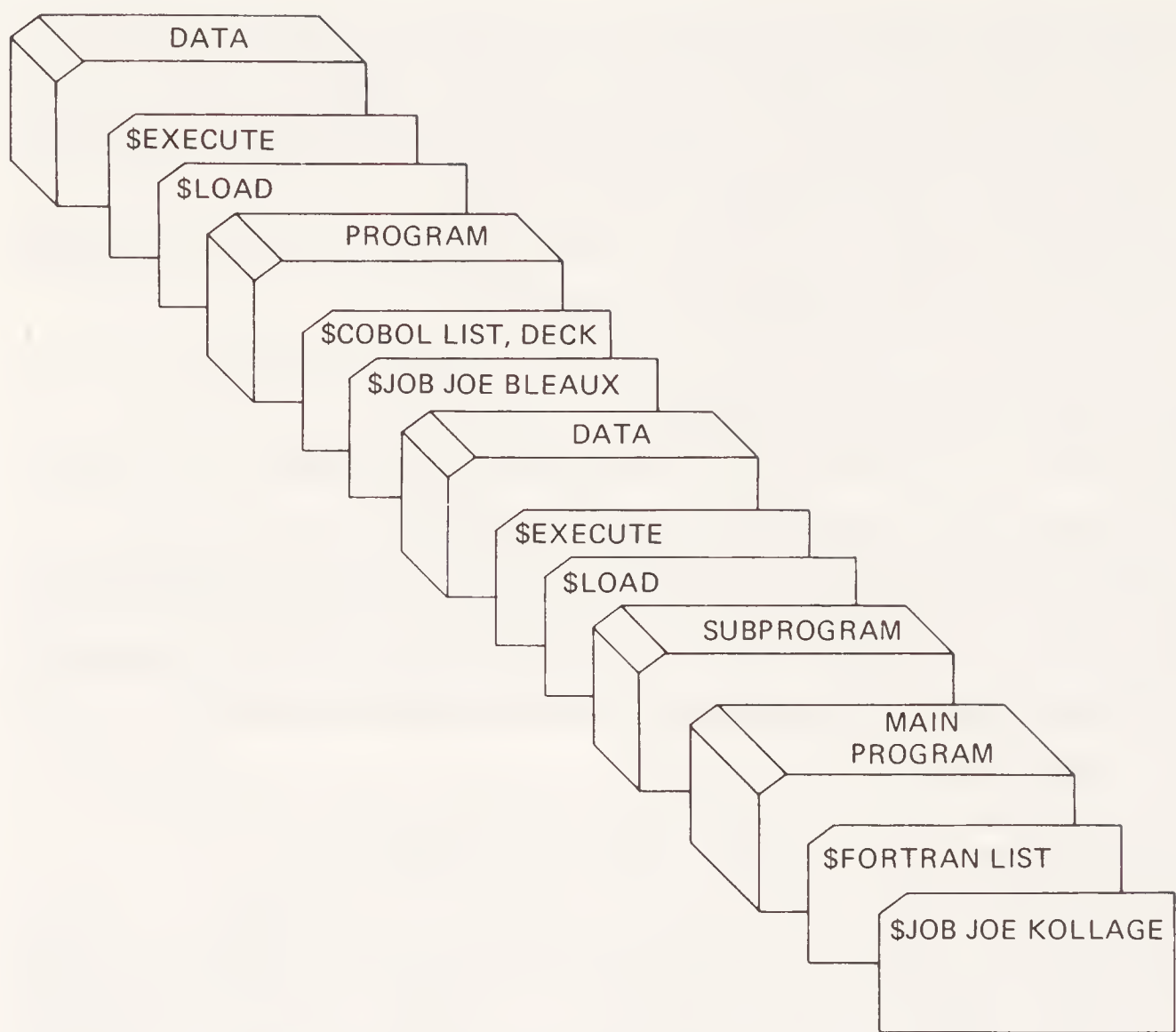


Figure 7-21 Job stream for a batch processor

that it would be ready for the next job. The current trend is to use a disk or drum for these purposes, since the desired compiler can be located much faster.

The second item that must be provided is an executive or monitor system that can supervise the operation of the entire process. This executive must be capable of processing a job stream that consists of a wide variety of programs, one after another. A typical input is illustrated in Figure 7-21. In this job stream, cards known as *control cards* are placed at key points to instruct the executive as to what tasks are to be performed.

In the job stream in Figure 7-21, the control cards all begin with the \$ character. The first card begins with \$JOB followed by the user's name. This signal to the executive is the beginning of a new "job." In response, the executive generally cleans away whatever remains from the previous job so that Joe Kollage is given a "clean slate." The next card begins with \$FORTRAN, and signals the executive that a Fortran program is to follow. The executive then locates the Fortran compiler on disk or tape and transfers it to memory. The \$FORTRAN card may also contain "parameters" such as LIST and NODECK. These specify which of various options are to be used by the compiler in processing the ensuing program. For example, LIST may specify that the source statements are to be listed on the output device; NOLIST would specify that the source statements are not to be listed. DECK might specify that an object

deck is to be punched after compilation is completed; NODECK might specify that no object deck is wanted. In the absence of a particular parameter, a “default” is assumed.

After the compiler is loaded, it then compiles whatever programs are provided. Some systems require additional control cards separating the main program and each of the subprograms; other systems do not. The object decks resulting from each compilation are written upon disk or tape for subsequent use by the loader.

The \$LOAD card signals the termination of the compilation phase and the beginning of the load phase. The loader is then located and transferred into memory. The object decks generated by the Fortran compiler are then transferred to memory by the loader, followed by the subroutine library. After loading is completed, the program is then executed.

At the termination of the execution phase, control is automatically transferred back to the executive. For example, the STOP statement in Fortran, the STOP RUN statement in Cobol, and similar statements in other programming languages are “stop” statements only in the sense that they terminate execution of the user’s program. Upon execution of any of these statements, the executive then takes over and proceeds to the next task to be done. In no sense does the machine cease to function.

In order to be effective, the executive must be capable of contending with virtually any abnormal condition that might occur during a run. For example, in the case of data for a particular program, the user might provide either too much data or not enough. In the case of too much data, the job is terminated with some data cards remaining to be processed. In this case, the executive must search through the job stream until the next control card is found. In the case of not enough data, the user’s program will read all data provided and then attempt to read the control card that follows. The executive must detect this as an erroneous condition, automatically terminate, or “abort,” the job in process, and proceed to perform the duties designated by the control card encountered. In this way, no user is able to interfere with the processing of the next user’s job.

Furthermore, some conditions that occur at one point in a run may make the performance of later requests impossible. For example, suppose the main program of Joe Kollage’s Fortran deck illustrated in Figure 7-21 contained serious errors in the source program, making it impossible for the Fortran compiler to generate an object deck. A “flag” is then set within the executive so that the load phase requested via the \$LOAD card will not be attempted. Then upon encounter of the next \$JOB card, this flag must be reset by the executive so as not to interfere with load requests by subsequent users.

Other extensions of the features presented in this section enable the executive to perform more effectively. For example, parameters on the \$JOB card might specify such things as a limit on the time permitted for the job, a limit on the pages or lines of output, and other similar controls. In all, the general intent is to permit the computer to “run itself” with a minimum amount of human intervention.

7.6. MULTIPROGRAMMING

While the batch processing approach is a considerable advance over manual operation, certain inefficiencies exist. When running programs with high volumes of input or output in the batch environment, the processor may be waiting on peripheral devices much of the time. This is circumvented to some extent by buffering peripheral devices, but this does not completely solve the problem. For example, the card reader may be provided with a buffer storage area where it can store the information read from a single card. If the processor wants to read a card, it actually reads this buffer area, which the card reader then proceeds to fill with the contents of the *next* card. In effect, the card reader is one card ahead of the processor. Suppose, however, that the processor wants to read two cards in rapid succession, as it would do when reading the source statements of a Fortran program. The first card is obtained from the buffer, but then the processor must wait until the next card has been entered into the buffer before “reading” it. At 600 cards/minute, this wait is only 0.1 second. But with a processor capable of a million operations per second, this wastes about 100,000 operations.

The objective of multiprogramming is to maintain two or more programs in memory simultaneously so that the processor can be working on one program while it is waiting for an input or output operation to be completed in the other program.

To operate efficiently in the multiprogramming mode, the processor must be able to run jobs in an order frequently different from that in which they were submitted. To do this, jobs are generally read from cards or other input medium onto disk or drum to form a “queue” of jobs to be run. The processor selects jobs from this queue as the necessary resources become available to run them. For example, if the processor only has 100K words available for all user programs, there is no way in which it can simultaneously place in memory two programs requiring 75K words each. It can, however, load one 75K word program and one 25K word program simultaneously.

Let's suppose that the processor has Program A and Program B in memory at this instant of time. As in the batch mode, the processor is executing only one job at a time but is capable of switching very quickly from one job to another. A common scheme is to do this switching when the program being run has an I/O request. For example, suppose the processor is executing Program A. Upon an occurrence of an I/O request by Program B, the processor initiates the request and immediately begins to work on Program B. Upon encounter of an I/O request by B, the processor resumes execution of Program A. In this way, the processor's time is shared between the two programs as illustrated in Figure 7-22. The allocation of the processor's time is governed by the occurrence of the I/O requests.

Allocation of memory may be either via fixed areas or via variable areas, each of which is illustrated in Figure 7-23. Early multiprogramming systems tended to use the fixed-partition arrangement in Figure 7-23a. Memory is divided into a specified number of areas or partitions, each of which could contain a single

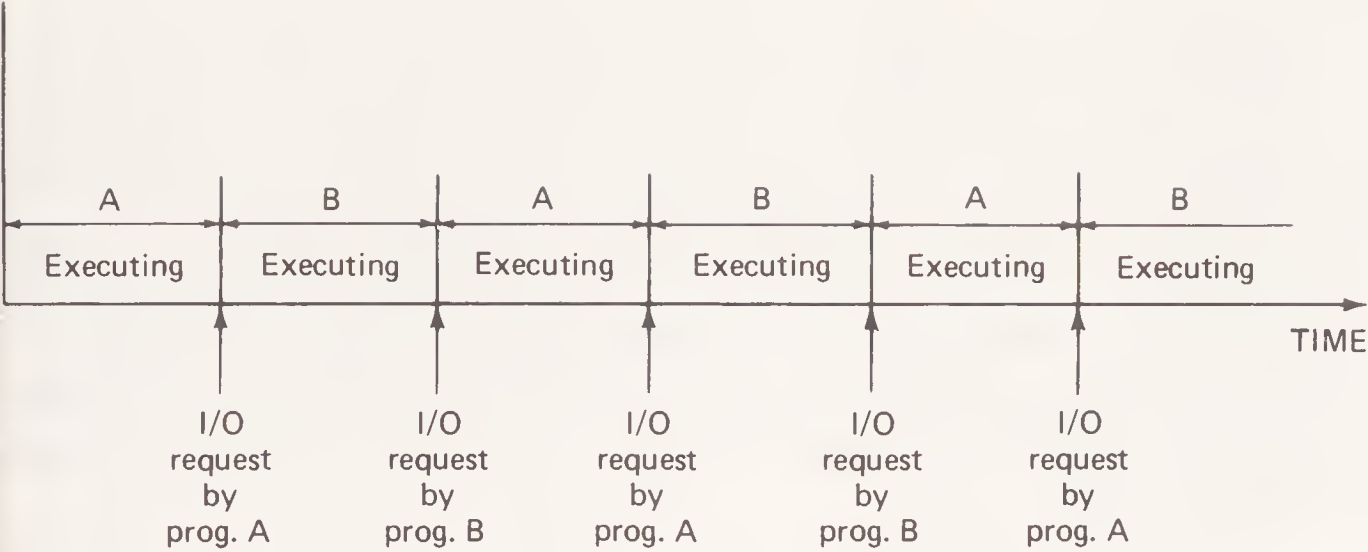
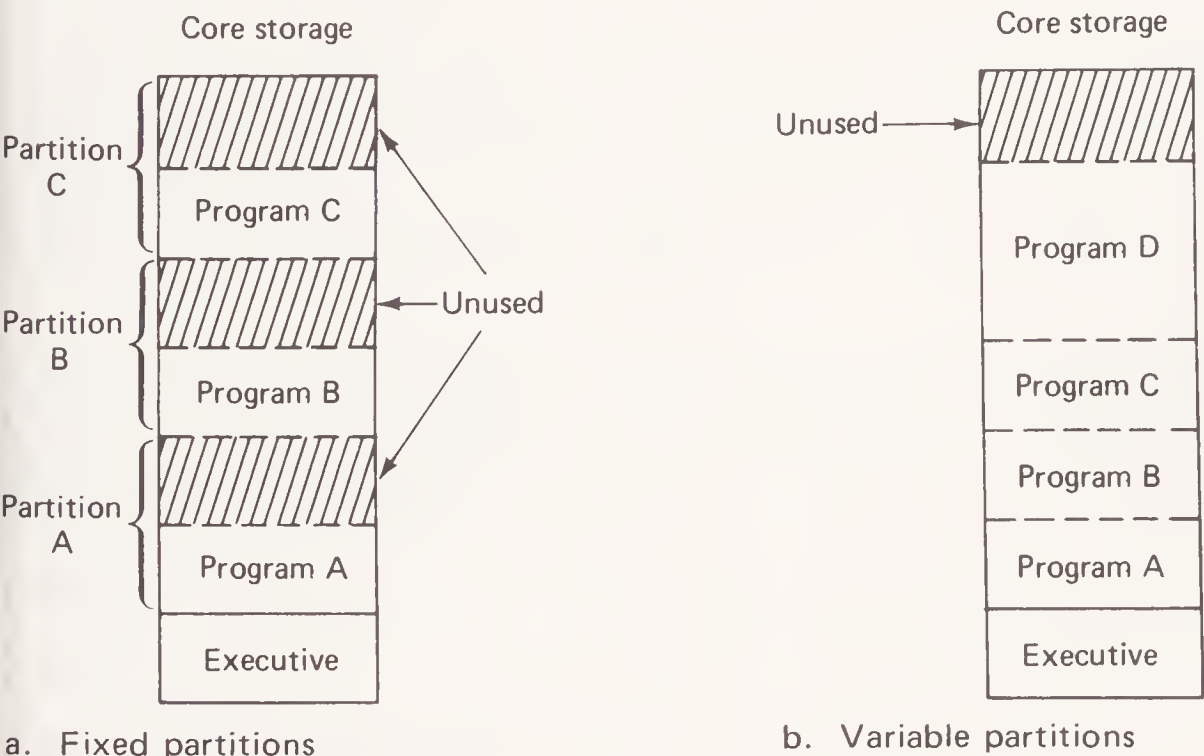


Figure 7-22 Time allocation in a multiprogramming environment

program. Loading of the programs always begins with the first location in the partition. Generally the memory is allocated to one large partition and two or more smaller ones.

The problem with the fixed-partition arrangement is that unused areas were interspersed between the various user programs. If these unused areas could be consolidated, another program could frequently be entered into memory as illustrated in Figure 7-23b. The concept behind the variable-partition arrangement is to adjust the size of the partition so that it exactly corresponds to the size of the user's program, thus eliminating the unused areas between programs. Loading of programs may now commence at any point in memory and is not restricted to the three or so points with the fixed partition arrangement illustrated in Figure 7-23a.

Although use of variable partitions utilizes memory more efficiently, a greater demand is made on the executive. For example, suppose Program B in Figure 7-23b is run to completion. Now the executive is faced with two alternatives.



a. Fixed partitions

b. Variable partitions

Figure 7-23 Allocation of memory

First, it could select a job from the waiting queue that would fit into the space vacated by Program B. If none were small enough, the space could be left vacant. In any case, unused areas of memory appear, much as in the case of the use of fixed partitions in Figure 7-23a. Alternatively, the executive could dynamically relocate programs C and D so that the unused space always appears at the upper section of memory. Which of these gives the best results depends upon the circumstances. In addition, some executives have the option of “rolling” a program out to disk and reloading it later to achieve more efficiency. Of course, it is conceivable that the overhead involved may more than offset any increased efficiency that results.

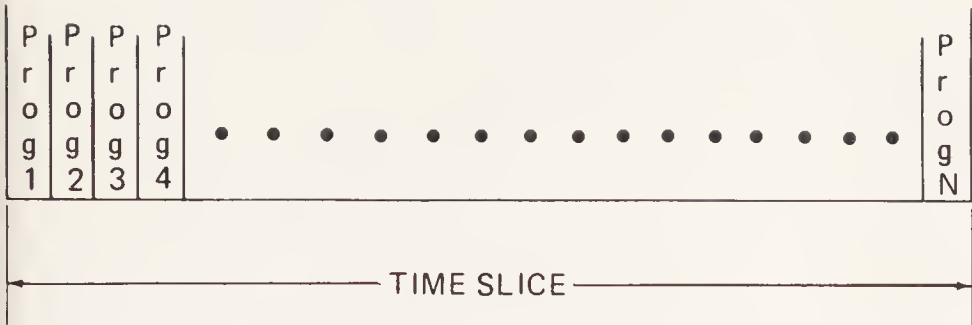
7.7. TIME-SHARING

To a certain extent, use of multiprogramming is really an extension of batch processing that permits the processor's resources to be used more fully. Programs are submitted as complete jobs in virtually the same fashion as a user submits a job to a conventional batch processor. He leaves his job to be run and returns in a few hours to pick up the results. While there is essentially no alternative for large programs, this arrangement is inconvenient for small ones. In effect, if the user could get a small fraction of the computer's resources over the next few minutes, he could get his program run. Time-sharing makes this available.

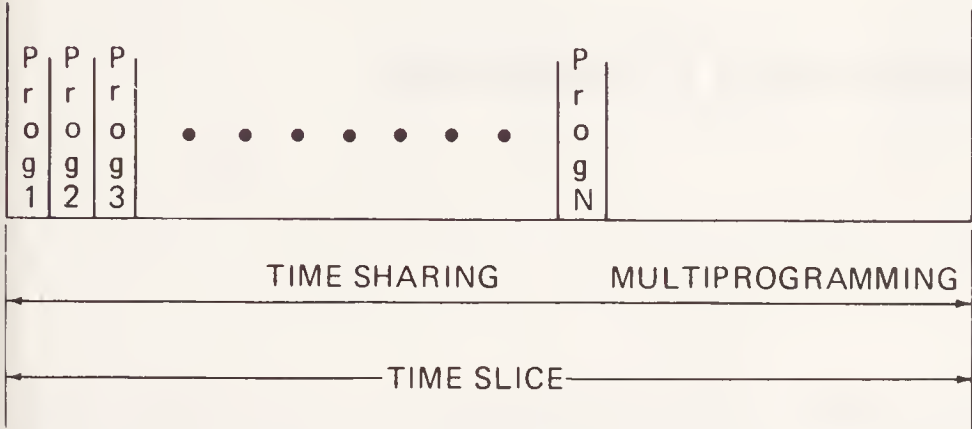
In this mode of operation, each user gets a certain fraction of the processor's attention over each small period of time known as a *time slice*. The time slice is further subdivided into smaller segments during which the processor is working upon a given user's program, as is illustrated in Figure 7-24a. Each program is run in turn for a small time (on the order of a hundred milliseconds) before the processor proceeds to the next program. In this way, each program is assured of getting its share of the processor's attention on a regular basis.

Since most time-sharing users will be communicating with the processor over a low-speed unit such as a teletype, the processor is fast enough to give each user the feeling that he has the computer all to himself. This mode of operation permits users to enter small programs, request that they be run, and obtain the results almost instantaneously, which is much superior to multiprogramming in that his program does not have to wait in the input queue until his turn comes around.

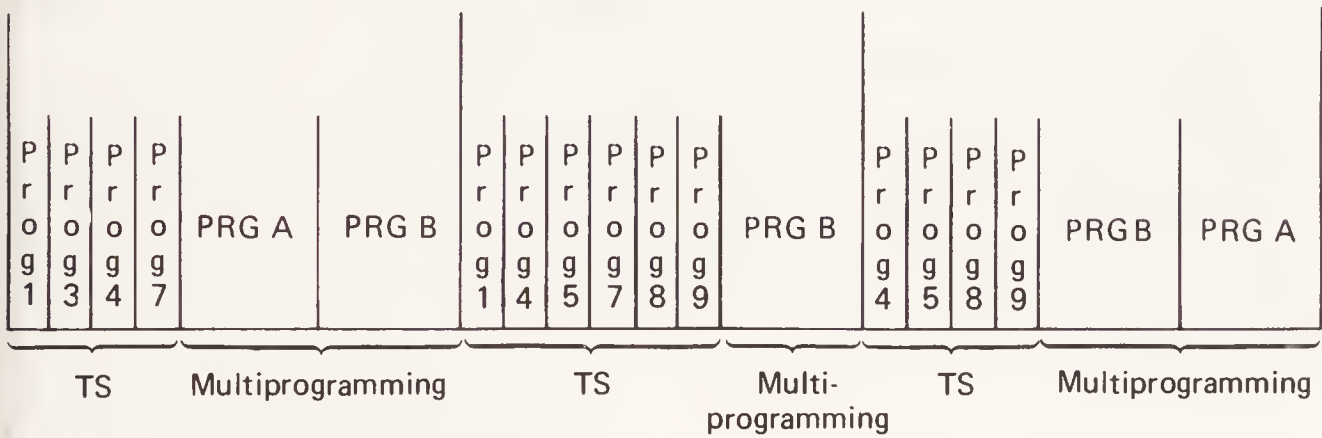
Many systems combine both time sharing and multiprogramming on the same machine. As illustrated in Figure 7-24b, some maximum portion of each time slice is devoted to time-sharing, that is, the case in which all programs are active over the time slice. Of course, not all time-sharing programs are generally active over every time slice, which causes the percent of time devoted to time-sharing to vary. In the example illustrated in Figure 7-24c, only time-sharing programs 1, 3, 4, and 7 are active over the first time slice, programs 1, 4, 5, 7, 8, and 9 are active over the next, and programs 4, 5, 8, and 9 are active over the last time



a. Division of time between N programs being run in time-sharing



b. Division of time between time-sharing and multiprogramming



c. Variation over successive time slices

Figure 7-24 Time-sharing

slice. Any user who is not computing during a particular time slice needs no time, even though his terminal may be connected to the machine. He may, for example, be keying in data or trying to diagnose the results of his last run and thus is not doing any actual computing.

The length of the time slice generally depends upon the patience of the users. After entering a request, a user may have to wait up to one time slice or slightly longer before his turn comes up again. Experience has shown that waits of 2 seconds or less are usually seen to be instantaneous, but waits of 5 seconds or more are rather annoying.

8

TREES, GAMES, AND NETWORKS

In previous chapters, we have studied the representation and processing of both numerical and nonnumerical information via the computer. In many problems, the basic information to be represented is that of *relationships*. The study of the representation of relationships begins with a study of trees, which is one of the most fundamental concepts in the field of computer science.

In this chapter, we shall first introduce the concept of a tree by illustrating its use to represent relationships between items of data. We shall then study the representation of the tree itself, followed by the use of trees in the analysis of games. Finally, we shall study a somewhat more complex type of relationship that is best represented by a network.

8.1. CONCEPT OF A TREE

The familiar *family tree* is a good starting point for the study of trees. Figure 8-1 illustrates two types of family trees—the pedigree and the lineal chart. The tree is said to consist of *nodes* and *branches*. In the trees of Figure 8-1, the nodes represent individuals, and the branches indicate parent-child relationships. The node at the top of each of the trees in Figure 8-1 is called the *root node*, or simply *root*. Although this is obviously contrary to the orientation of natural trees, we shall generally represent our trees in this upside-down configuration.

Should the root node be deleted from either of the trees in Figure 8-1, we would be left with two *subtrees* in Figure 8-1a and four subtrees in Figure 8-1b. The subtree has all of the properties of a tree, which enables a tree to be defined in a *recursive* manner, that is, defined in terms of itself, as illustrated by the following definition: A *tree* is a finite set of one or more nodes with the following properties:

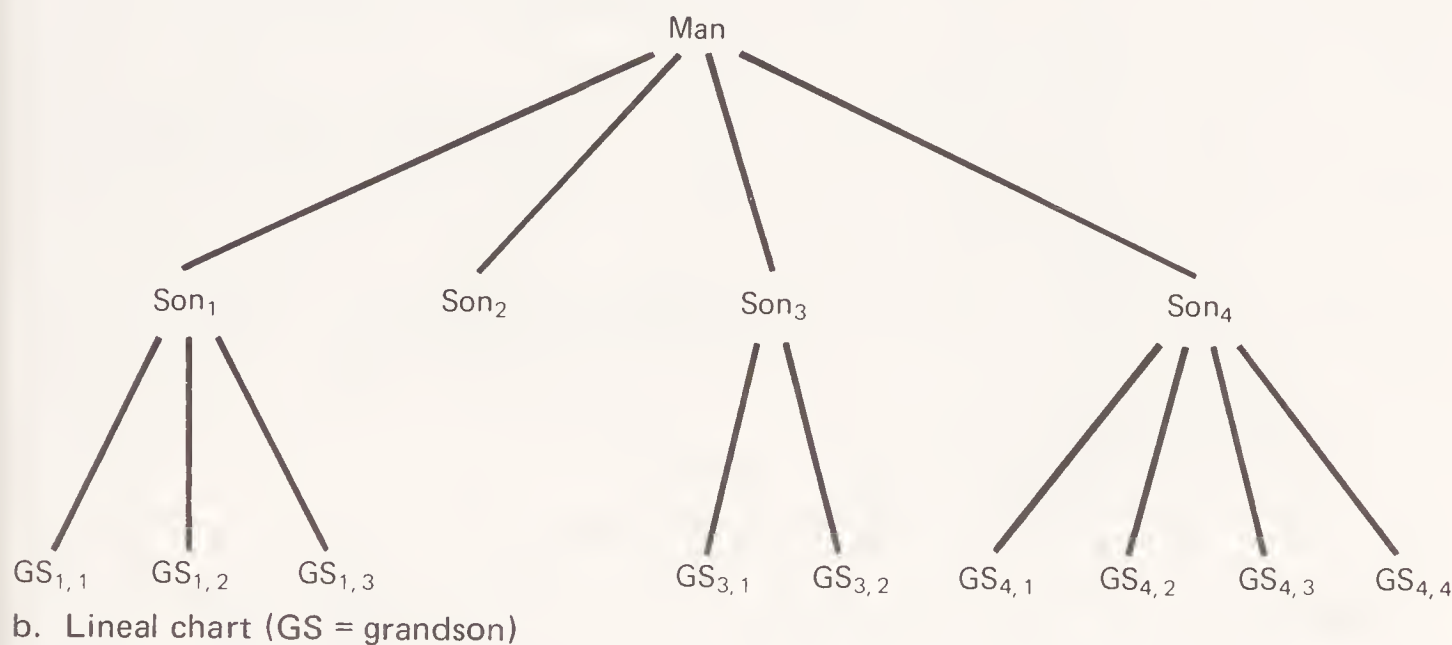
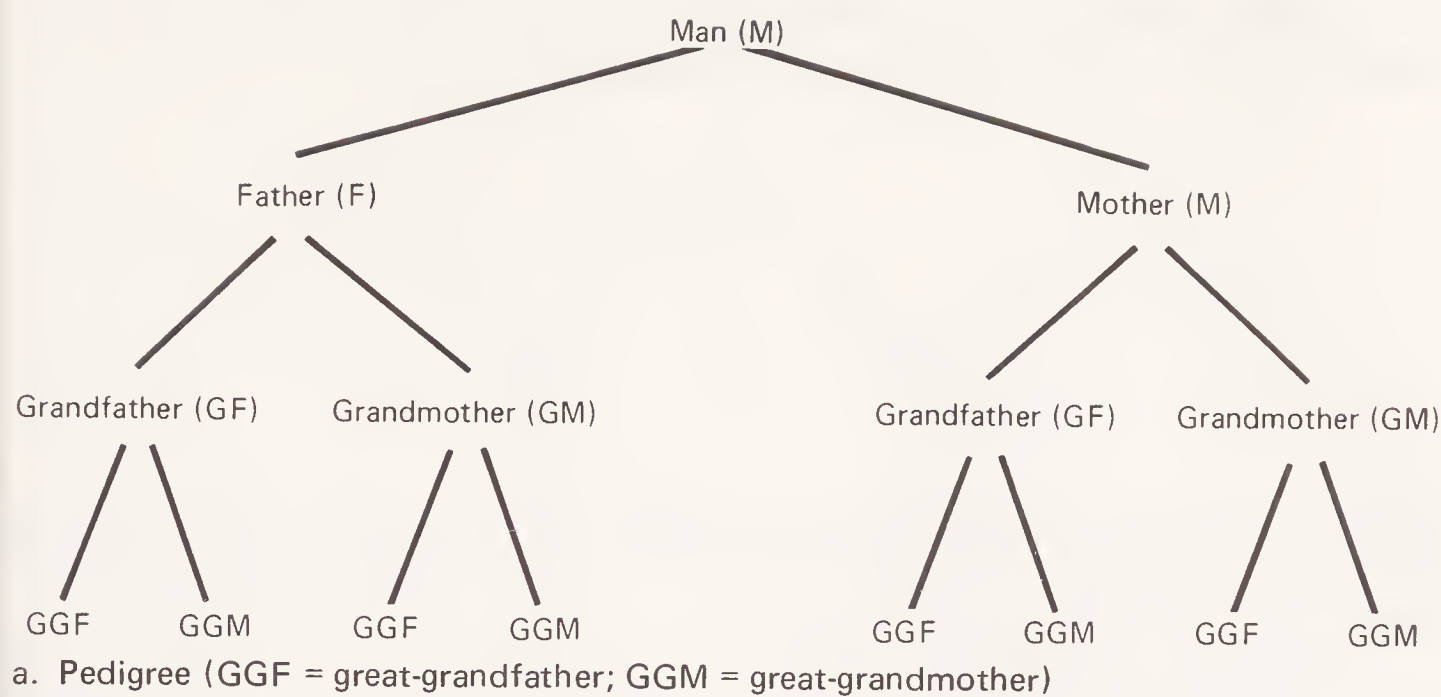


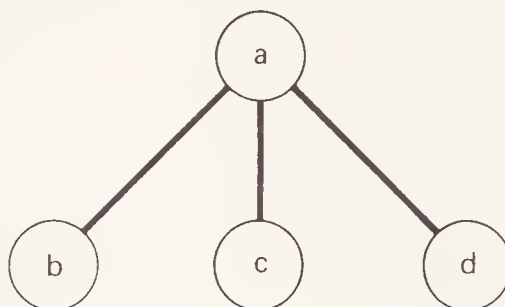
Figure 8-1 Family tree

- 1. There is one specially designated node called the *root* of the tree.
- 2. If the root node is excluded, the remaining nodes (if any) form a set of subtrees in which no node occurs in more than one subtree and where each subtree has the same properties as the tree itself.

One consequence of this definition is that a tree may consist of a single node. Furthermore, no branches may “overlap”; that is, there must be one and only one path from the root node to each of the nodes in the tree itself.

For example, overlapping would occur in the pedigree type of tree in Figure 8-1a whenever cousins marry. That is, should first cousins marry, they would have a common grandparent. Although this complication can be circumvented by defining the nodes as representing not an individual but “an individual in the parent role,” we shall simply ignore this complication in our subsequent discussion.

A certain amount of the terminology used with trees is derived from the family tree. For example, consider the following tree:



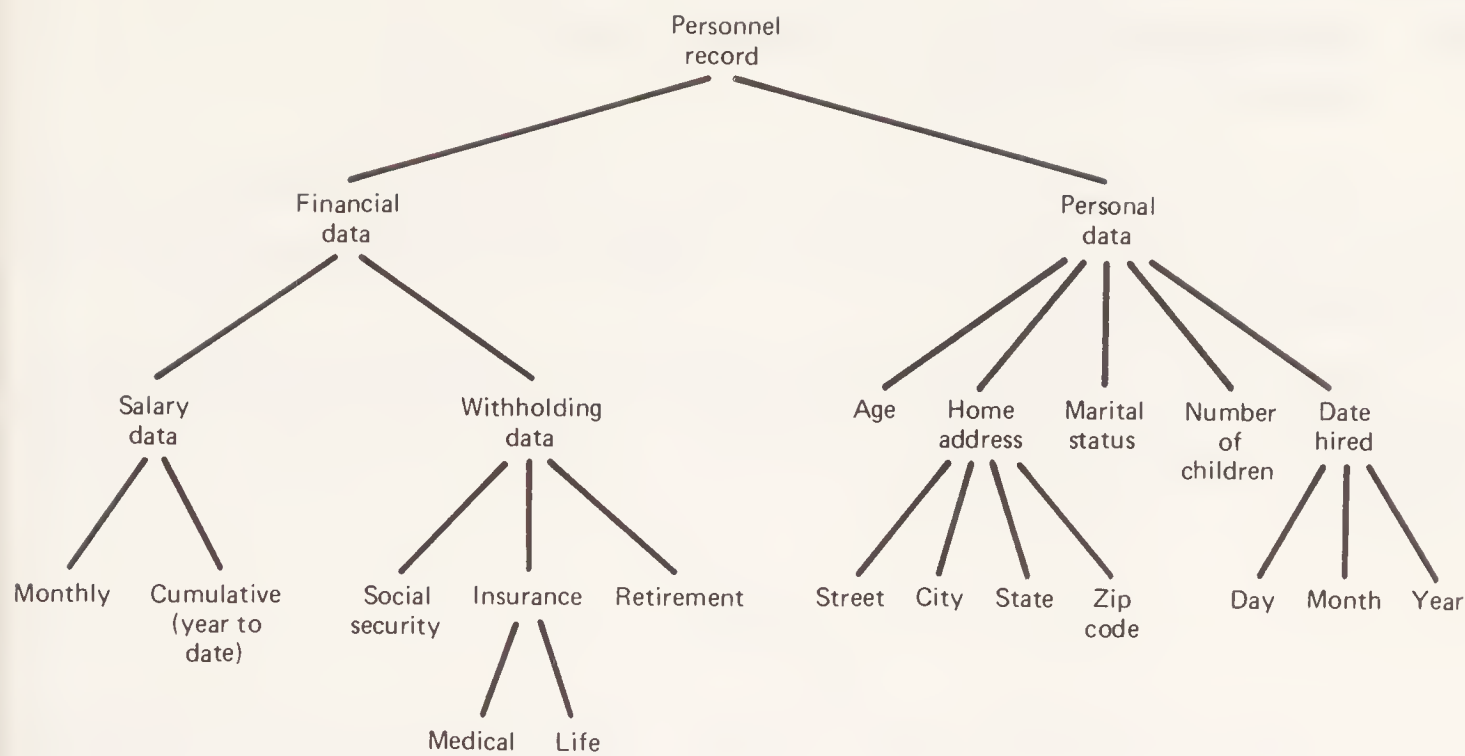
Node *a* is referred to as the *parent* node, and nodes *b*, *c*, and *d* are referred to as *offspring*. Furthermore, nodes *b*, *c*, and *d* are brothers or sisters. A node with no offspring is said to be a *terminal node*.

In Chapter 4, we defined a node in terms of information structures as being the basic element of a linear list and containing in turn more elemental forms of information. In most cases, a definite relationship exists between the information in the node, which introduces a role for a tree. In PL/I and Cobol, a node of this type is generally referred to as a *record*. To avoid confusion in this section, we shall use the term record to mean a node in information structures context and use the term node only in its tree context.

As an example of a record, we might consider the data on Joe Bleaux in our employee records, which contains the following information:

1. Current monthly salary
2. Cumulative salary paid to date this year
3. Social Security withheld to date this year
4. Monthly payment for group medical insurance
5. Monthly payment for group life insurance
6. Monthly contribution to retirement fund
7. Age of employee
8. Street on which employee resides
9. City in which employee resides
10. State in which employee resides
11. Zip code for mailing purposes
12. Marital status of employee
13. Number of children
14. Day of month in which employee was hired
15. Month in which employee was hired
16. Year in which employee was hired

The tree representation in Figure 8-2a is very useful in that it presents the relationships between the individual items of information in addition to specifying what items are contained in the record. In programming languages such as Cobol or PL/I, this structure is included in the definition of the record, as illustrated in Figure 8-2b.



a. Tree representation of personnel record

```
01 PERSONNEL-RECORD.
  02 FINANCIAL-DATA.
    03 SALARY-DATA.
      04 MONTHLY          PIC 9(4)V99.
      04 CUMULATIVE       PIC 9(6)V99.
    03 WITHHOLDING-DATA.
      04 SOCIAL-SECURITY  PIC 9(3)V99.
      04 INSURANCE.
        05 MEDICAL        PIC 9(3)V99.
        05 LIFE           PIC 9(3)V99.
      04 RETIREMENT       PIC 9(3)V99.
  02 PERSONAL-DATA.
    03 AGE                PIC 9(3).
    03 HOME-ADDRESS.
      04 STREET           PIC X(20).
      04 CITY             PIC X(20).
      04 STATE            PIC X(4).
      04 ZIP-CODE         PIC 9(5).
    03 MARITAL-STATUS     PIC 9.
    03 NUMBER-OF-CHILDREN PIC 99.
    03 DATE-HIRED.
      04 DAY              PIC 99.
      04 MONTH            PIC 99.
      04 YEAR             PIC 9(4).
```

b. Representation in Cobol

Figure 8-2 Structure of the data in personnel record

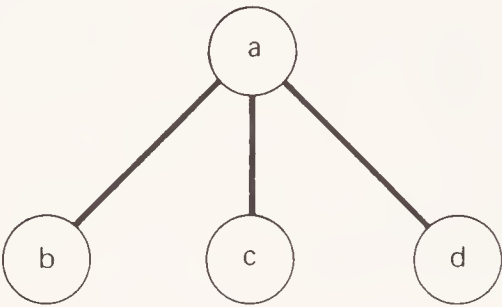
In effect, the tree in Figure 8-2a has been represented in the Cobol program segment in Figure 8-2b. At this point, we shall examine the representation of trees in more detail. Later in this chapter, we shall examine the use of trees in the analysis of games. In the next section we shall use trees to represent expressions.

268

8.2. REPRESENTATION, STORAGE, AND
SEARCH OF TREES

As a basis for the discussion in this section, we shall use from Figure 8-2 the subtree whose root is “Financial Data.” Abbreviating the names at each node yields the tree in Figure 8-3.

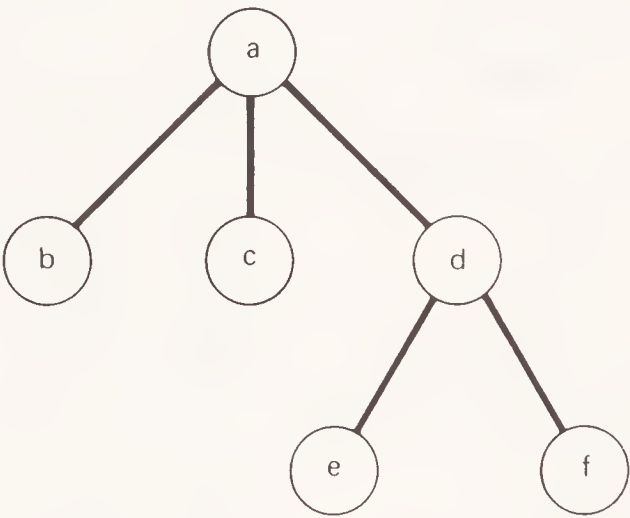
A more concise, nonpictorial representation of a tree can be obtained by use of a list. Although a variety of notational practices could be followed, one possibility is to represent the tree



as

$$L = a:(b,c,d)$$

The labels of the branch nodes are enclosed in parentheses, and the label of the root node is attached with a colon. Use of nesting permits this type of notation to be extended to more complex trees such as



$$L = a:(b,c,d:(e,f))$$

The list for the tree in Figure 8-3 is

$$L = FD: (SD:(MO,CU),WD:(SS,IN:(MD,LI),RE))$$

While lists can be used to represent trees, they are not synonymous. For example, the list for the tree in Figure 8-3 contains the sublists

$$SD:(MO,CU)$$

and

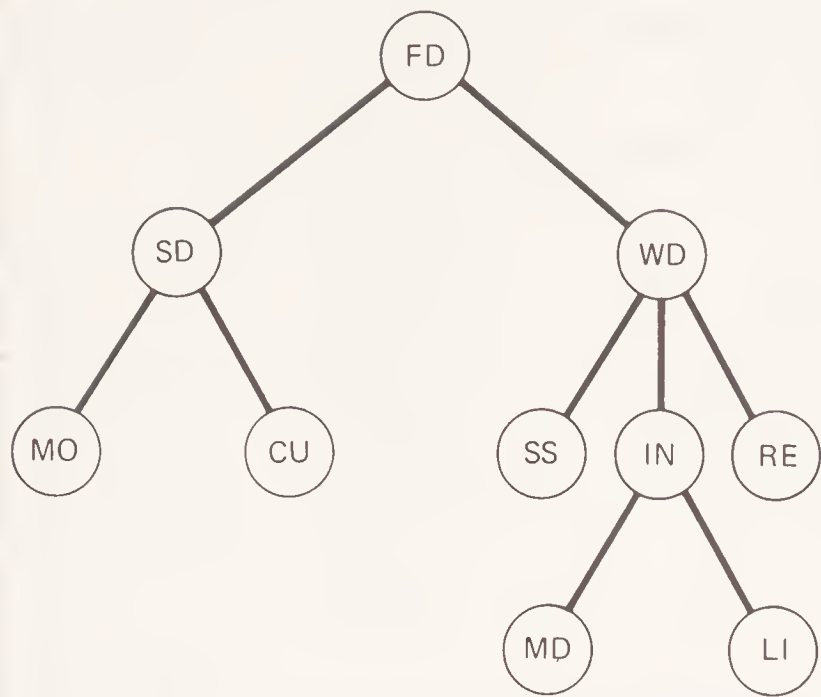


Figure 8-3 Subtree with “Financial Data” as root node from Figure 8-2

WD:(SS,IN:(MD,LI),RE)

In general, such sublists are not necessarily disjoint; that is, they may overlap. In these cases, there would be no tree corresponding to the list.

One approach to representing trees in a manner suitable for storing in the computer’s memory is to use a double-subscripted array. Numbering the nodes as in Figure 8-4 would permit the storage of the terminal nodes of each branch, illustrated as follows:

1	2
2	4
2	5
1	3
3	6
3	7
7	9
7	10
3	8

The first column represents the parent node; the second column represents one of its offspring. Each row corresponds to a branch in the tree.

Some observations can be made about this representation. For example, all nodes except the root node appear once and only once in the second column. Furthermore, any node not appearing in the first column is a terminal node. However, certain shortcomings beset this representation. For example, the route from node 1 to node 10 can be determined only by several scans of the array, although a systematic organization can lead to better efficiency.

An alternative approach is to use a square array, that is, a double-subscripted array with an equal number of rows and columns. Rows correspond to parent nodes and columns to offspring. For example, if node 6 is an offspring of the

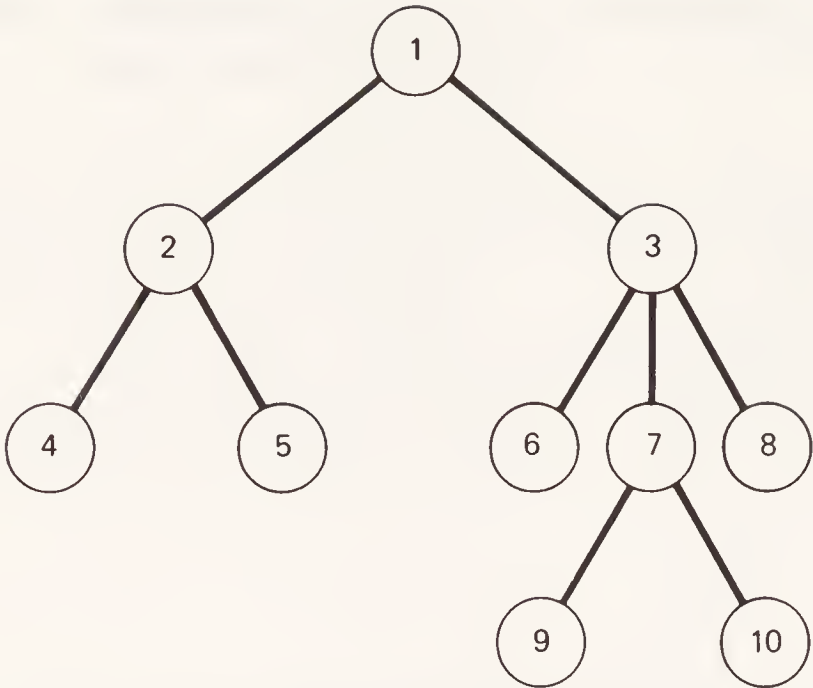


Figure 8-4 Numerical designation of nodes of tree in Figure 8-3

parent node 2, a 1 is placed in row 2, column 6 of the array; otherwise, a 0 is placed in this location. Using this convention, the tree in Figure 8-4 could be represented as follows:

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	1	1	1	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

The problem with this representation should be obvious: too many zeroes are stored. Using a hundred storage locations to represent a tree with ten nodes is not very efficient.

Many applications require a storage representation that permits one to easily proceed from a terminal node to the root node or from the root node to a terminal node. Use of the one-way list presented in Chapter 4 permits this requirement to be met.

We shall first consider a representation that will permit us to readily proceed from a terminal node to a root node. To permit this, for each node we shall store the following:

- 1. The label of the node itself.
- 2. The label of its parent node.

This is illustrated by the representation in Figure 8-5. Our one-way list would be as follows:

FD , * (* denotes the root node)
SD , FD
MO, SD
CU , SD
WD, FD
SS ,WD
IN ,WD
MD, IN
LI ,IN
RE , WD

Alternatively, using the numbered representation in Figure 8-4 gives the following list:

1, 0 (0 denotes the root node)
2, 1
3, 1
4, 2
5, 2
6, 3
7, 3
8, 3
9, 7
10, 7

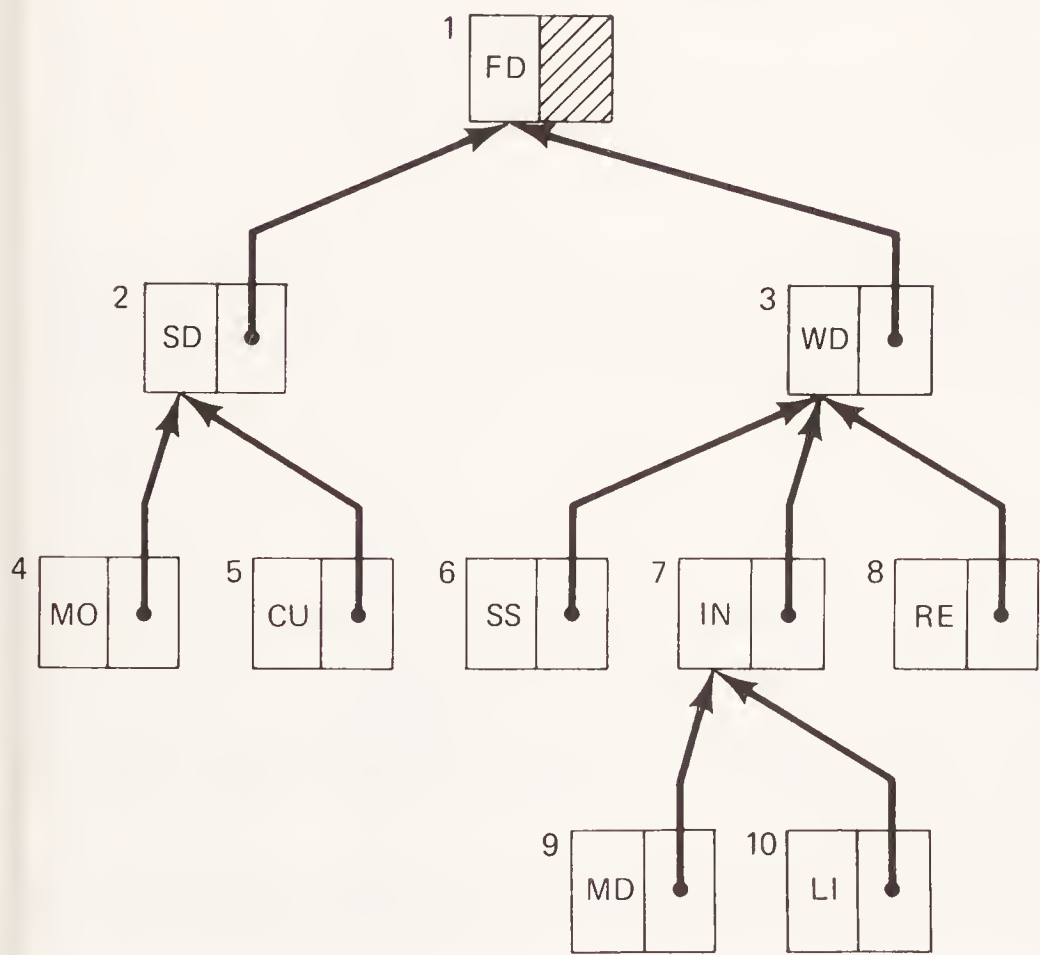


Figure 8-5 Representation of tree structure via a one-way list

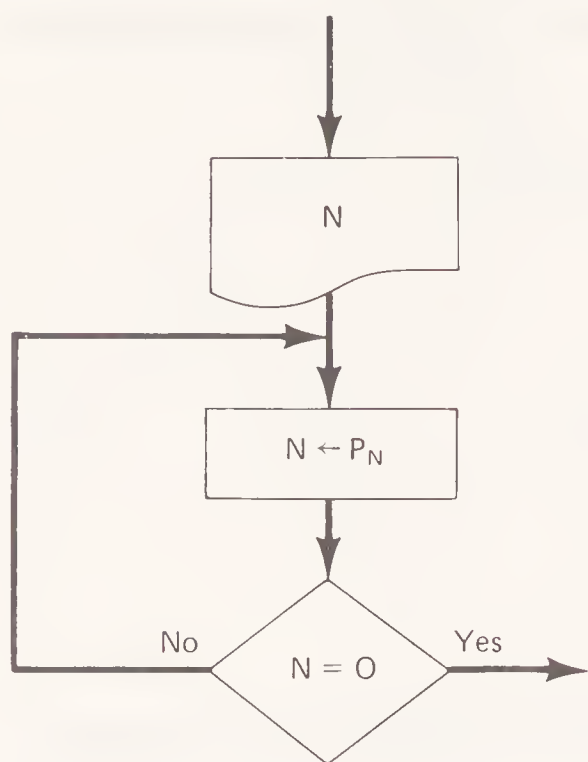


Figure 8-6 Algorithm for traversing from a terminal node to the root node

This representation is very similar to one presented earlier except that an entry is included for the root node and the columns are reversed. Since, however, the first column is numbered in sequence, it need not be stored. If the second column is stored in the single-subscripted array P in the order specified by the first column, $P_6 \leftarrow 3$, for example, the algorithm for proceeding from node N to the root node is given in Figure 8-6.

In other cases, one needs to traverse from the root node to the other nodes in the tree, which is not very convenient using the representation in Figure 8-5. The natural first suggestion is to “reverse” the arrows, giving the representation in Figure 8-7. For this case, the list might appear as follows:

```
FD ,SD ,WD ,*
SD ,MO,CU ,*
MO,* ,* ,*
CU ,* ,* ,*
WD,SS ,IN ,RE
SS ,* ,* ,*
IN ,MD,LI ,*
MD,* ,* ,*
LI ,* ,* ,*
RE,* ,* ,*
```

Each entry consists of the label of the parent node followed by the labels of all of its offspring. The $*$ indicates the absence of an offspring node.

In the above list, three positions are provided for offspring nodes. Since only one of the nodes, namely, node **WD**, has three offspring, this third position is “wasted” for all of the other nodes.

This waste can be avoided by arranging the arrows as shown in Figure 8-8. In

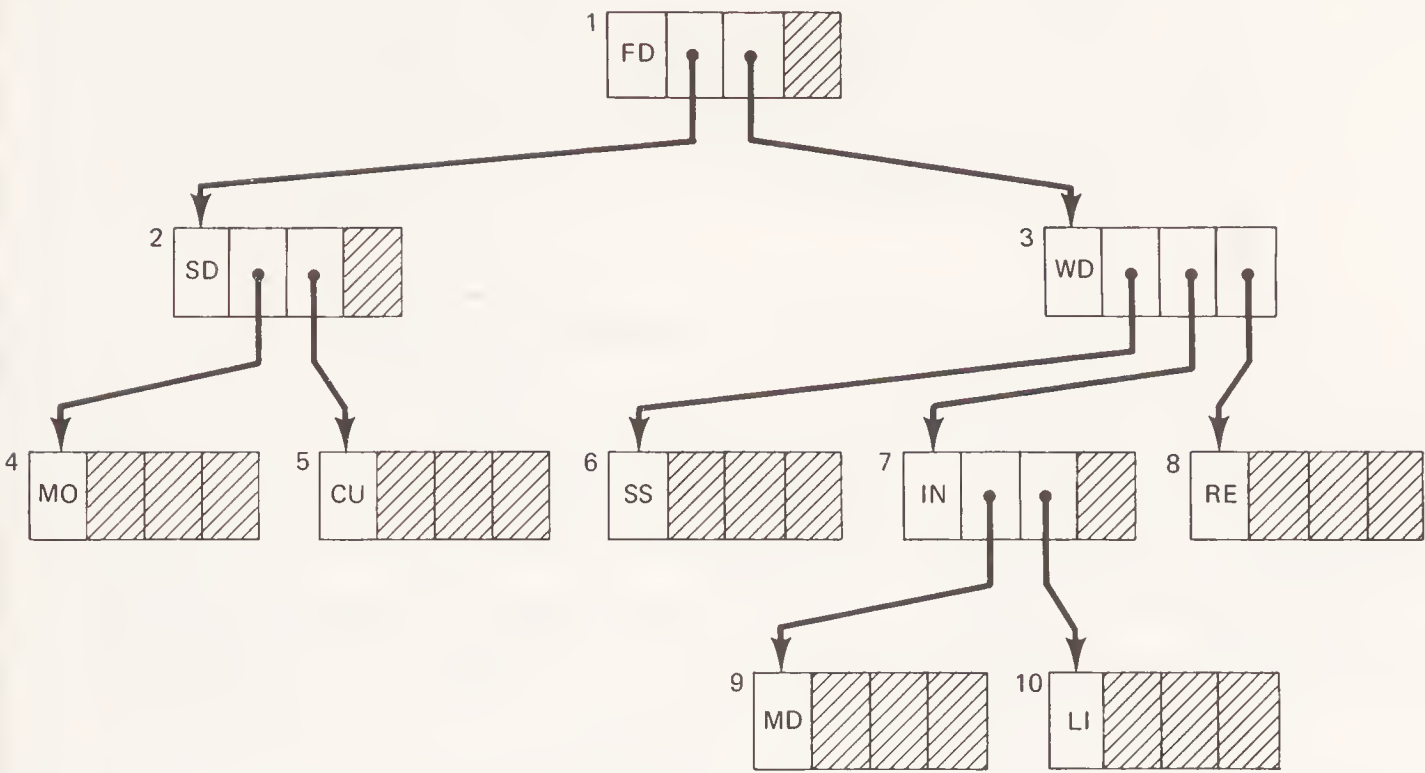


Figure 8-7 Alternative representation of tree

this representation, two entries are associated with each node. The first entry points to the parent node's first offspring; the second entry points to the *parent node's* next brother or sister. In this way, only two entries are associated with each node, giving the following list:

FD ,SD ,*
SD ,MO,WD
MO,* ,CU
CU ,* ,*
WD,SS ,*
SS ,* ,IN
IN ,MD,RE
MD,* ,LI
LI ,* ,*
RE ,* ,*

Alternatively, using the numbered notation gives the following list (intentionally arranged in ascending order of the parent node's number):

1, 2, 0
2, 4, 3
3, 6, 0
4, 0, 5
5, 0, 0
6, 0, 7
7, 9, 8
8, 0, 0
9, 0, 10
10, 0, 0

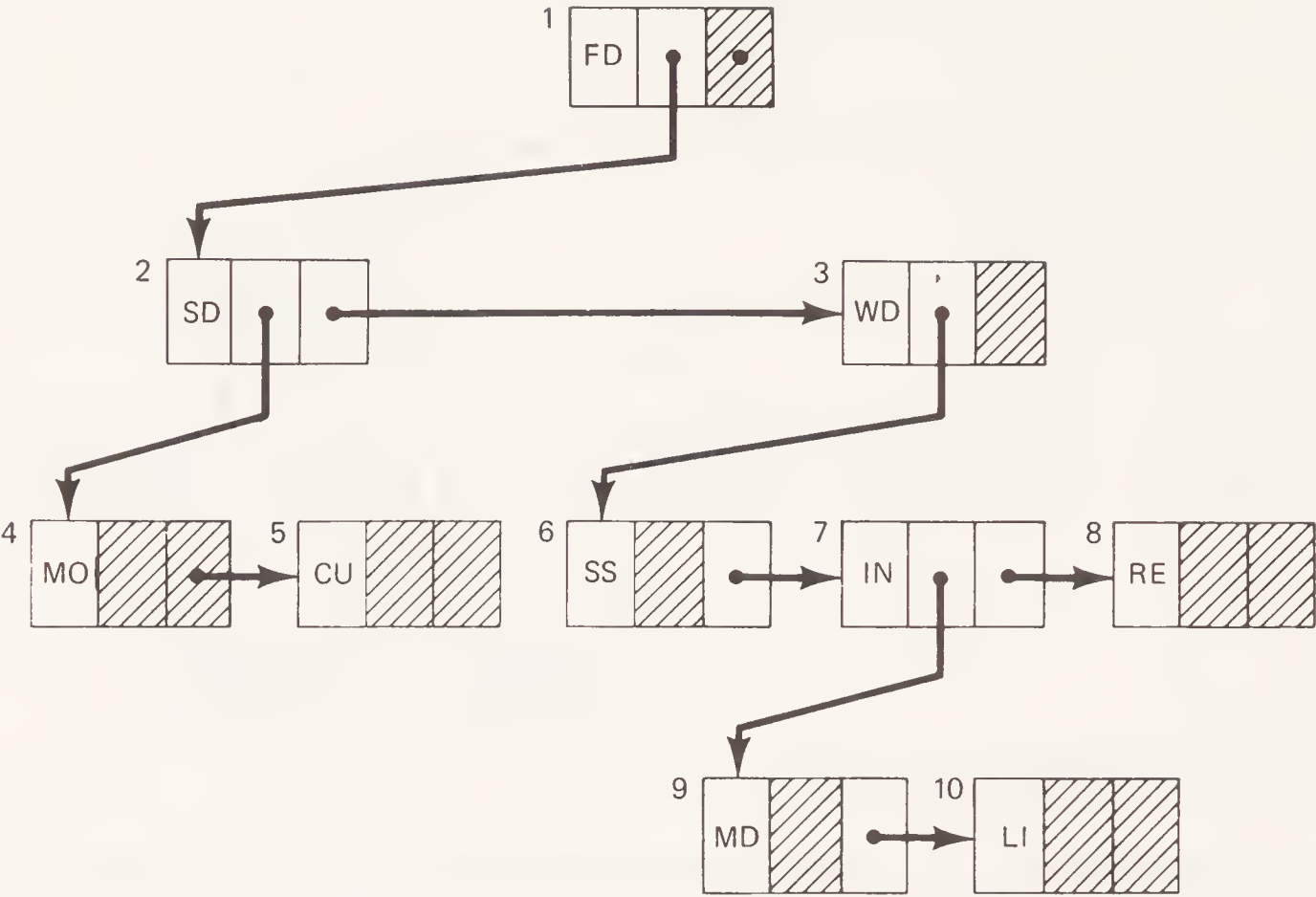


Figure 8-8 Tree representation using only two locations per node

Since the parent nodes appear in ascending order, this representation can be stored using two single-subscripted arrays, *C* for the “child” or second column and *B* for the “brother” or third column. These arrays are defined as follows for this example:

$$C = 2, 4, 6, 0, 0, 0, 9, 0, 0, 0$$
$$B = 0, 3, 0, 5, 0, 7, 8, 0, 10, 0$$

One situation in which this representation is quite convenient is for searching the tree, that is, examining each node in search of some condition much as a squirrel would search a natural tree for a nut. In our search, suppose we are to initially examine the root node and then examine each of the subtrees, beginning with the leftmost subtree. In examining each subtree, we first examine the root node and then examine each subtree of the subtree. In effect, we have recursively defined our search procedure. Referring to Figure 8-3, we visit nodes as follows:

FD, SD, MO, CU, WD, SS, IN, MD, LI, RE

In this way, we are searching the tree in much the same manner as a squirrel would search a natural tree; and hence the term *natural tree search*. The steps are as follows:

1. Begin with the root node FD
2. Visit the left subtree, beginning with its root node SD
3. Visit the node MO
4. Visit the node CU

5. Return to the right subtree, beginning with its root node WD
6. Visit the node SS
7. Visit the subtree whose root node is IN, starting with node IN
8. Visit node MD
9. Visit node LI
10. Visit node RE

This completes the search.

Using the arrays C and B defined earlier, the flowchart in Figure 8-9 represents an algorithm for visiting the nodes in this order. The flowchart also uses an additional array STK that functions as a “stack.” The logic is as follows:

1. Print the number of the node.
2. If there is an offspring node ($C_J \neq 0$) and a brother or sister node ($B_J \neq 0$), insert B_J into the stack and proceed to node C_J in step 1.
3. If there is an offspring node ($C_J \neq 0$) but no brother or sister node ($B_J = 0$), proceed directly to node C_J in step 1.
4. If there is no offspring node ($C_J = 0$) but there is a brother or sister node ($B_J \neq 0$), proceed directly to node B_J in step 1.
5. If there is neither an offspring node ($C_J = 0$) nor a brother or sister node ($B_J = 0$), proceed to the node at the top of the stack.

When the condition in step 5 occurs and the stack is empty, the tree has been searched.

8.3. A GAME

One popular pastime for computer “nuts” is to spend hours “teaching,” that is, programming, a computer to play games such as tic-tac-toe, checkers, and chess. In effect, a game consists of a sequence of decisions alternating between the two (or perhaps more) players. Since the computer is quite capable of making decisions very rapidly even when taking into account rather large amounts of data, it would be expected that a properly programmed computer would be an expert player of many games.

While devoting attention to games might seem to be a waste of valuable time, the pertinent concepts are applicable to many other aspects of our society. In effect, the functioning of our business activity is much like a game. First, your opponent makes a move, for example, addition to his product at no extra cost of a what-ever-it-is that can be obtained in a wide variety of colors. Now it is your move, and you might respond by reducing the price of yours. Now it is his move.

The complicating factor regarding real-world games is that they are not generally played by a fixed set of rules, which makes games like tic-tac-toe much more attractive for study in a text like this. However, the rules of some real-world games can be specified reasonably well if a degree of uncertainty is included to give more realism or to take care of the chap who does not “play by the rules.”

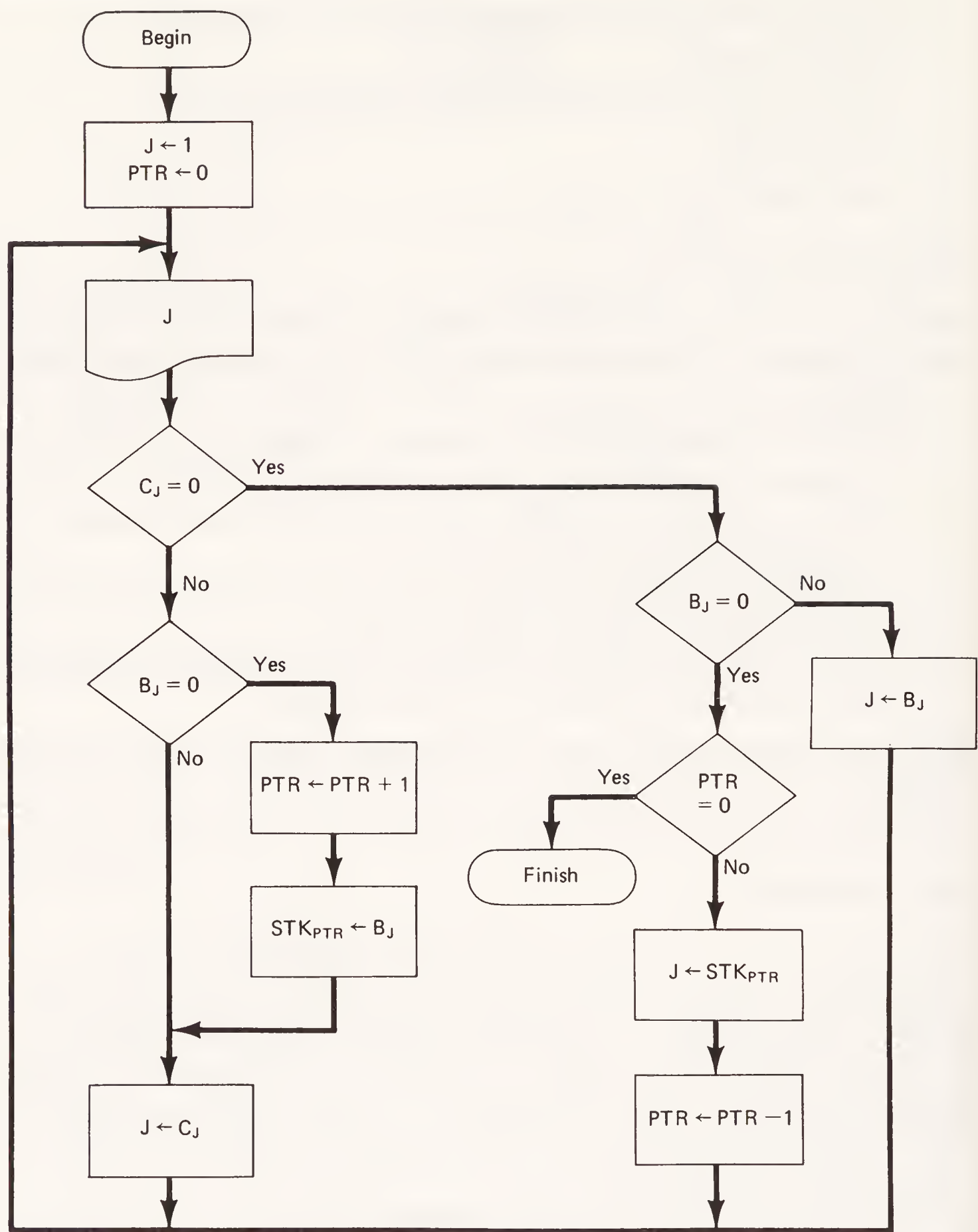


Figure 8-9 Algorithm for natural tree search

In the analysis of many games, the concept of a tree can be very useful. The nodes of the tree denote decision points, and the limbs represent the various paths that can be followed. We shall begin our discussion by illustrating how a tree can be used to represent the game of tic-tac-toe.

Although it is practically inconceivable that anyone would not be familiar with the game of tic-tac-toe, we shall nevertheless review the rules that we will follow. The game is played on a pattern similar to a number sign:

Each player alternatively places his symbol in one of the positions. Typical symbols used are X and O. The game continues until one player has three of his symbols in a line horizontally, vertically, or diagonally. In the following example, X wins:

X	X	X
	X	O
O		O

If neither player has three symbols in a row when all positions are filled, the game ends in a draw. The objective of the game is preferably to win, but if this is not possible, to at least obtain a draw.

To assist in the ensuing discussion, we shall number the positions as follows:

1	2	3
4	5	6
7	8	9

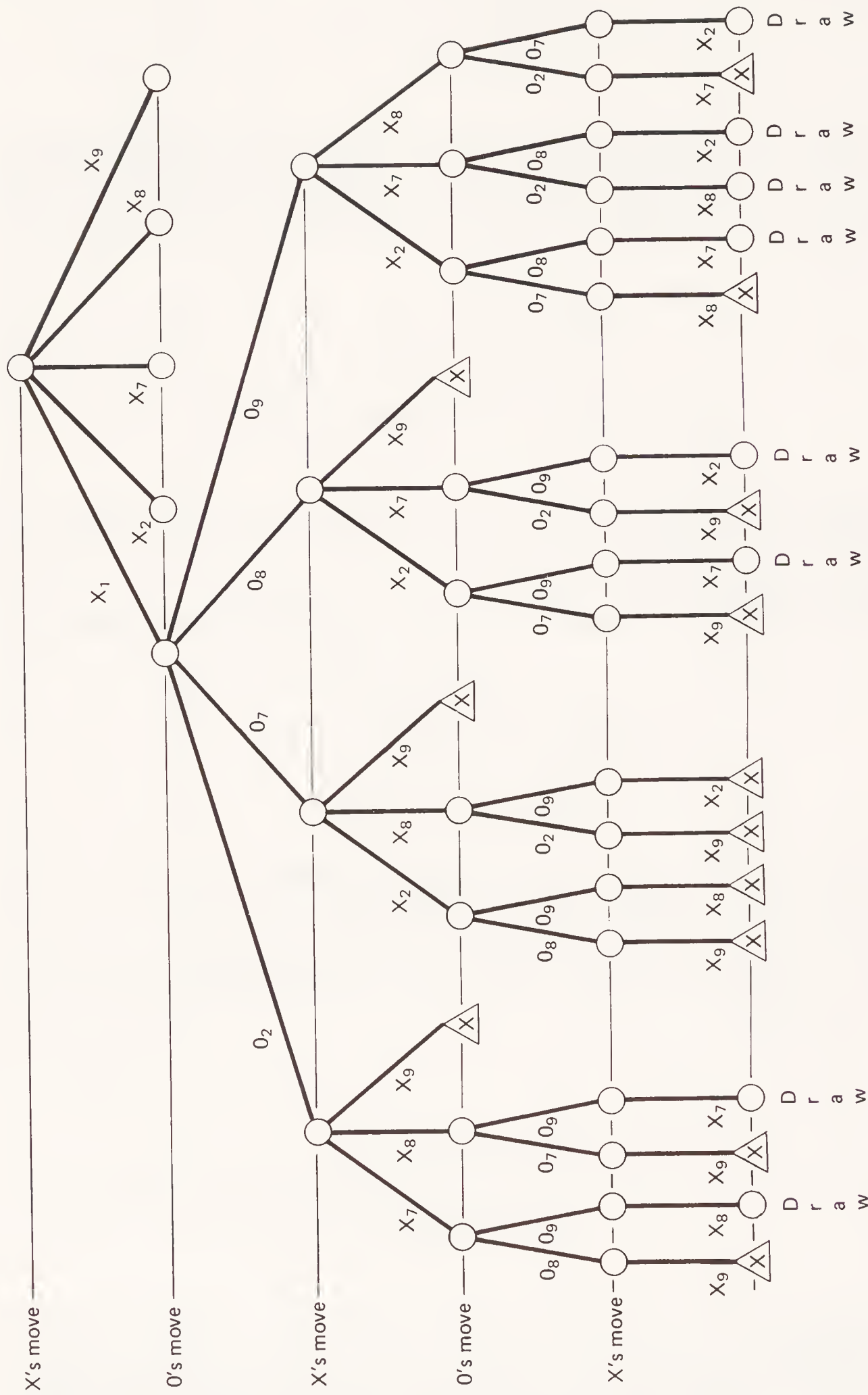
These numbers will be used as subscripts to designate the position of the symbols. For example, an X in position 2 would be denoted as X_2 .

In order to keep our tree down to a manageable size, let us develop a tree that will finish a partially complete game. Specifically, suppose the current status of the game is as follows:

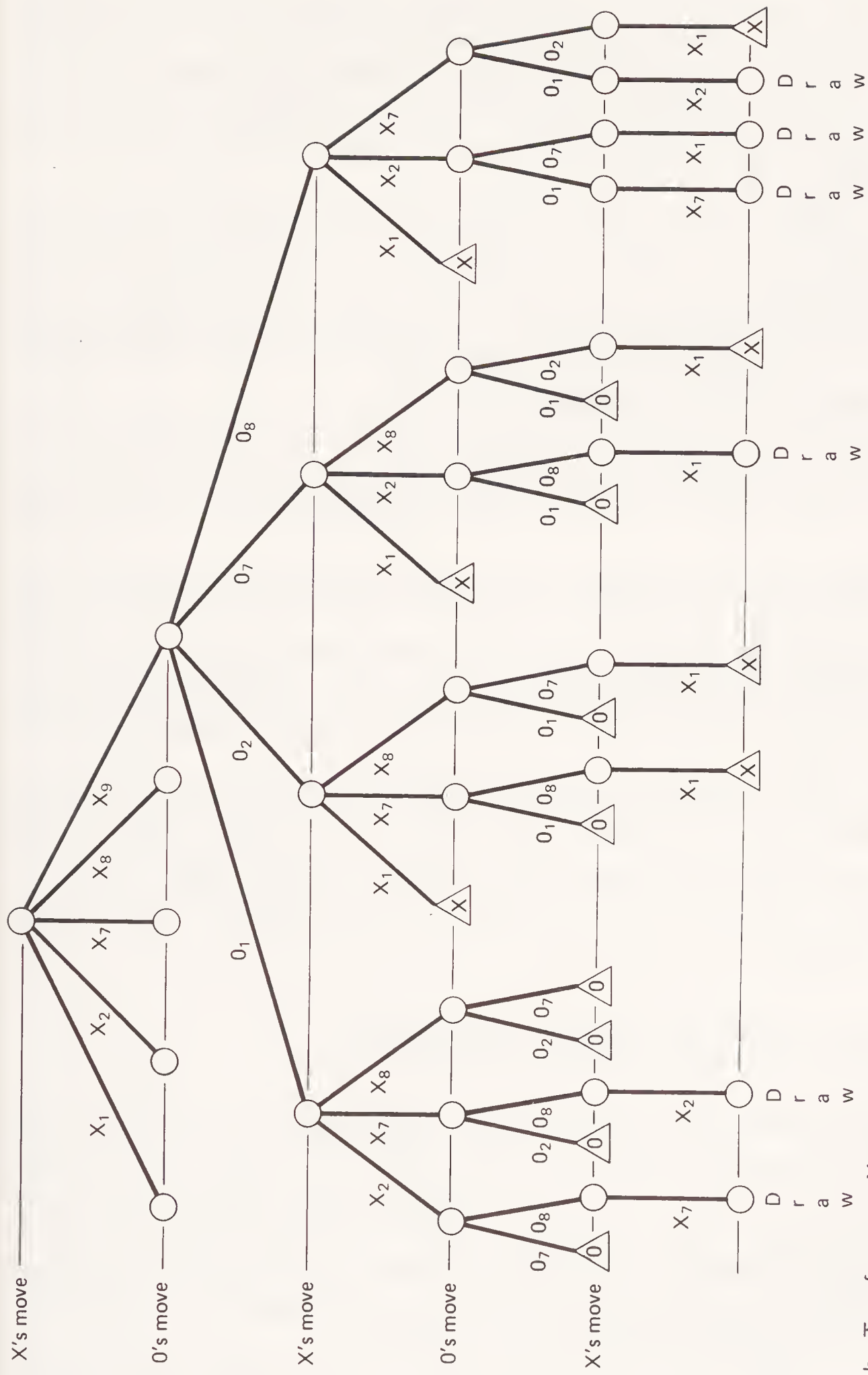
1	2	3
		O
4	5	6
O	X	X
7	8	9

Now it is X's move.

Mr. X has five possible moves: X_1 , X_2 , X_7 , X_8 , or X_9 . While he cannot win directly, is there a move that he can make so that, regardless of what his opponent does, he will eventually win the game? If so, he wants to make it. If there is no such move, is there a move that would enable his opponent to win, regardless of Mr. X's future moves? If so, Mr. X wants to avoid it. In making



a. Tree for move X_1
Figure 8-10 Tree representation of tic-tac-toe (triangular boxes denote a win for one of the players)



b. Tree for move X_9
Figure 8-10 (Continued)

these decisions, Mr. X will assume that this opponent is an “expert” player and will always make the best move.

The possible paths that Mr. X can follow from this point on are illustrated by the tree in Figure 8-10. The upper node illustrates that five paths are open to Mr. X, none of which will directly result in victory. After Mr. X makes his choice, Mr. O will have four possibilities open to him. Owing to space limitations, the tree in Figure 8-10a has been completed only for the move X_1 and the tree in Figure 8-10b only for move X_9 .

If Mr. X selects move X_1 , Mr. O can make moves of either O_2 , O_7 , O_8 , or O_9 . Moves of either O_2 , O_7 , or O_8 would permit Mr. X to win the game on the next move. But if Mr. O chooses move O_9 , the game continues, and Mr. X must choose either X_2 , X_7 , or X_8 . No matter which move Mr. X chooses, there is some move by which Mr. O can force the game to end in a draw. Therefore, if Mr. X chooses move X_1 , the game will at least terminate in a draw.

Now suppose Mr. X chooses move X_9 . Mr. O will then be given the choice of moves O_1 , O_2 , O_7 , and O_8 . If O_1 is chosen, then no matter what move Mr. X makes next, the game will be won by Mr. O. Therefore, Mr. X should avoid move X_9 .

Completing the tree in Figure 8-10 would reveal that Mr. X could at least obtain a draw from moves X_2 , X_7 , and X_8 , but none would insure a win against an expert Mr. O. Thus, Mr. X could choose from moves X_1 , X_2 , X_7 , or X_8 . To aid him in his choice, he might select the one that gives Mr. O the maximum chance of making an error, but we will not continue our problem beyond this point.

In games such as this, the computer is capable of scanning the tree to select the move which is most beneficial. As the game becomes more complex and the number of possible paths increases even more, the computer is able to take into account more alternatives than most human players and thus becomes an expert player of the game.

8.4. PROBLEMS INVOLVING DECISIONS

A well-known problem involving multiple decisions goes something like this: Suppose you are given eight balls, seven of which are known to be of equal weight and one known to be equal to or heavier in weight than the other seven. To locate the heavy ball, you may use only an equal arm balance, which can compare only the weights of whatever is placed on the two arms and indicate which is heavier. In the usual formulation, the objective is to make this determination with the minimum number of weighings.

Using the decision tree in Figure 8-11 will locate the ball in two weighings. The first step is to weigh any three of the balls against any other three. If either arm is heavy, then the heavy ball must be one of the three on the heavy arm. It can be located by weighing one of the three balls against any other one of the

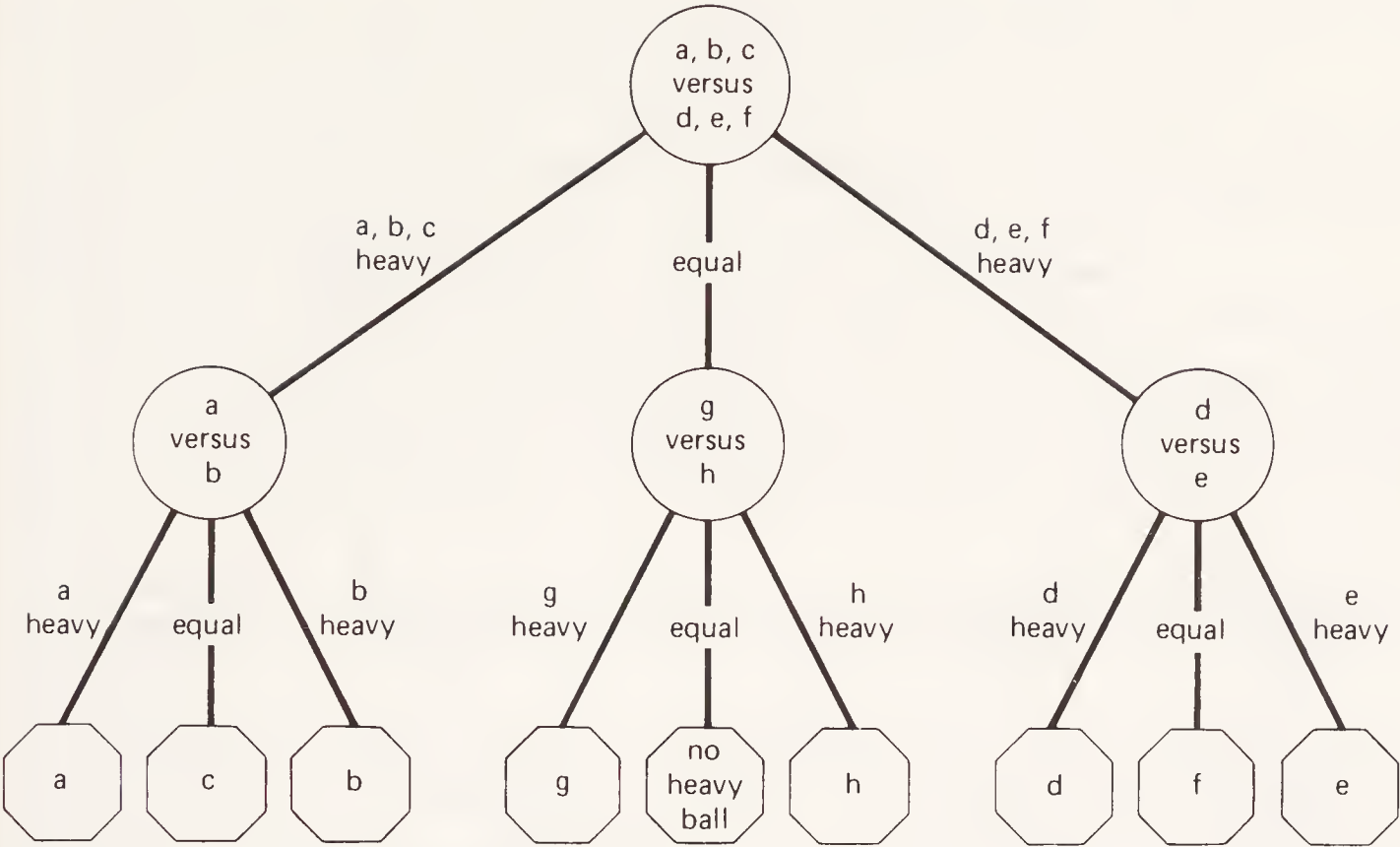


Figure 8-11 Tree for locating heavy ball

three. If the two selected balls are of equal weight, the heavy ball must be the third.

If the two groups of three balls are of equal weight, then the heavy ball (if there is one) must be one of the two balls not included in either group of three. Weighing these two balls gives the answer.

In problems of this type, one advantage of the tree is its explicit nature. The procedure to be followed is far better explained by the tree in Figure 8-11 than by the previous discussion.

In the previous case, the objective was to locate the heavy ball in a single sample in the minimum number of weighings. Let's change the problem just slightly. Suppose we have a thousand samples in which to locate the heavy ball in each of the samples. Utilizing the procedure illustrated in Figure 8-11, two weighings would be required for each sample, or a total of 2,000.

Now suppose we look at the records of past samples and discover that, on the average, only 10 percent of the samples actually contained a heavy ball, and we have no reason to believe that our thousand samples will be significantly different. Can we use this characteristic to design a procedure for locating the heavy balls with fewer total weighings?

Let's examine the tree in Figure 8-12. If one of the balls is heavy, three weighings are required to locate it. But if there is no heavy ball, this will be concluded with only one weighing. It is this characteristic which will be useful in our problem. The total number of expected weighings is as follows:

(1000 balls) × (10% with one heavy) × (3 weighings) = 300

(1000 balls) × (90% with all equal) × (1 weighing) = 900

Total expected weighings

1,200

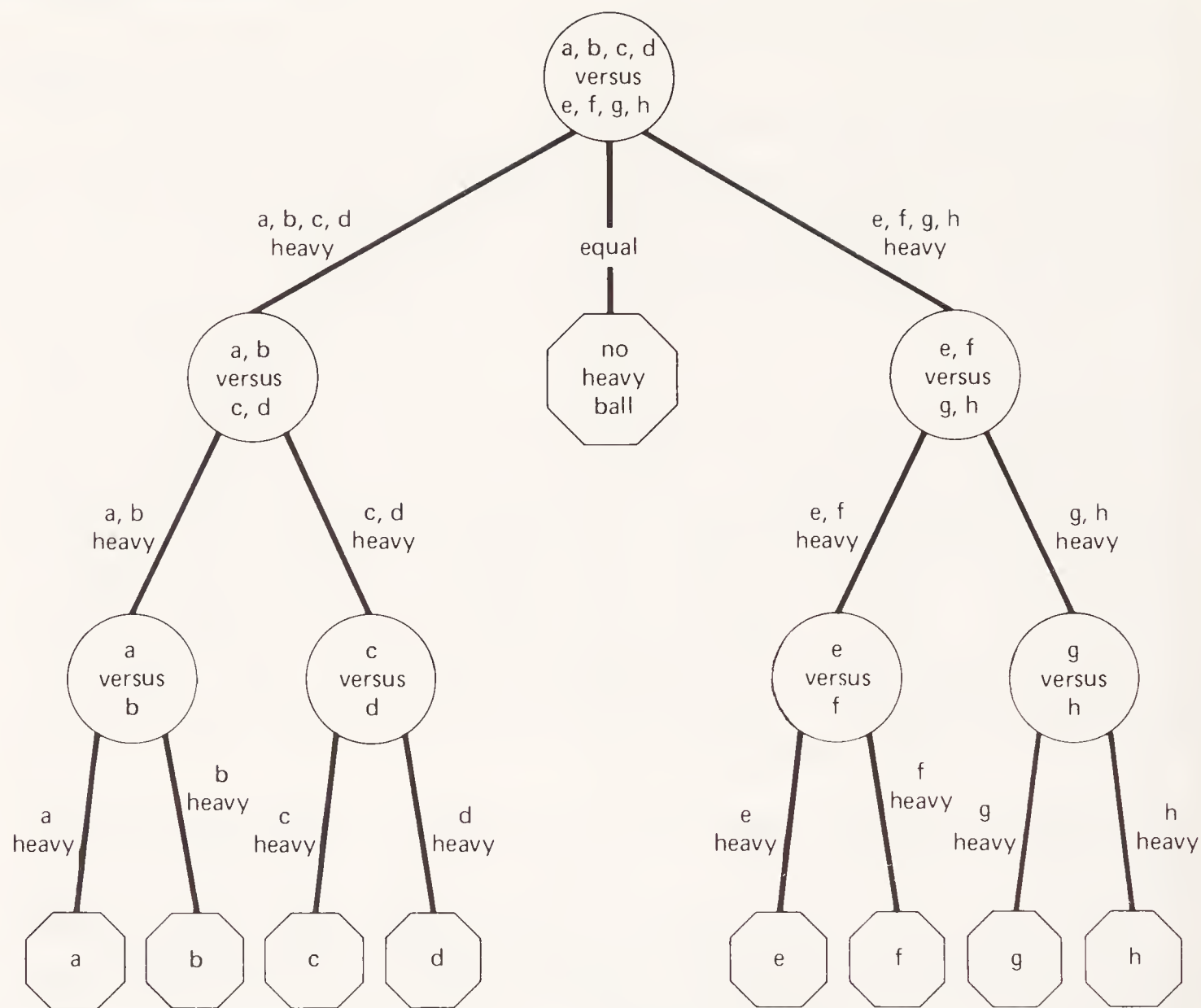


Figure 8-12 Alternative tree for locating the heavy ball

This is considerably fewer than the 2,000 weighings required by the scheme in Figure 8-11.

Note, however, the use of the term “expected” weighings. Depending upon the accuracy of the 10 percent figure, the actual number of weighings may range from 1,000 to 3,000. That is, if no sample contains a heavy ball, 1,000 weighings will suffice. If all samples contain a heavy ball, then 3,000 weighings will be required. Unless the 10 percent figure is accurately known, the exact number of weighings cannot be precisely computed in advance using the scheme in Figure 8-12. This uncertainty is so characteristic of many of the problems encountered in the real world.

Actually, the method in Figure 8-12 can be improved slightly, but we shall leave this aspect to the reader to pursue on his own.

8.5. A TRANSPORTATION PROBLEM

Suppose a traveler wants to go from point *A* in Figure 8-13 to any one of the three points *B*. The possible routes between these points and the cost of each are

given in the figure. The problem is to select the appropriate route so that the traveler goes from point A to any one of the points B at minimum cost.

At the starting node A and at each of the intermediate nodes 1 and 2, the traveler must decide which one of two possible paths he shall take. We shall designate these two paths as “up” and “down.” For example, if the traveler chooses up from point A , the trip to node 1 costs him 30 units.

Although the diagram in Figure 8-13 bears some resemblance to a tree, it does not meet at least one of the criteria, namely, that subtrees are disjoint, that is, have no nodes in common. Considering A as the root node, its two subtrees with root nodes 1 and 2, respectively, are not disjoint, since node B_2 is common to all of them.

Representation of networks in memory is somewhat analogous to storage of trees, although complicated by the overlapping. We shall consider this in the exercises.

To work this problem, it is best to essentially start at B and “back up.” For example, suppose we ask the question, what is the proper route and cost to go from point 1 to the final destination? An inspection of Figure 8-13 reveals that the proper choice is up at a cost of 30 units. Similarly, the best path to use from node 2 to the final destination is the down path at a cost of 20 units.

Now for the question of the best route from node A to the final destination. If our traveler calls up at node A , it costs him 30 units to get to node 1 plus another 30 units to get to his final destination, for a total cost of 60 units. Alternatively, if he chooses to call down at node A , it costs him 20 units to get to node 2 plus another 20 units to reach one of the B nodes, or a total of 40 units. Thus, we conclude that the best decision at node A is to call down followed by another call down at the intermediate node 2.

In working this problem, we have assumed that no matter how our traveler reaches a given node, he takes the best or minimum cost path from there to his destination.

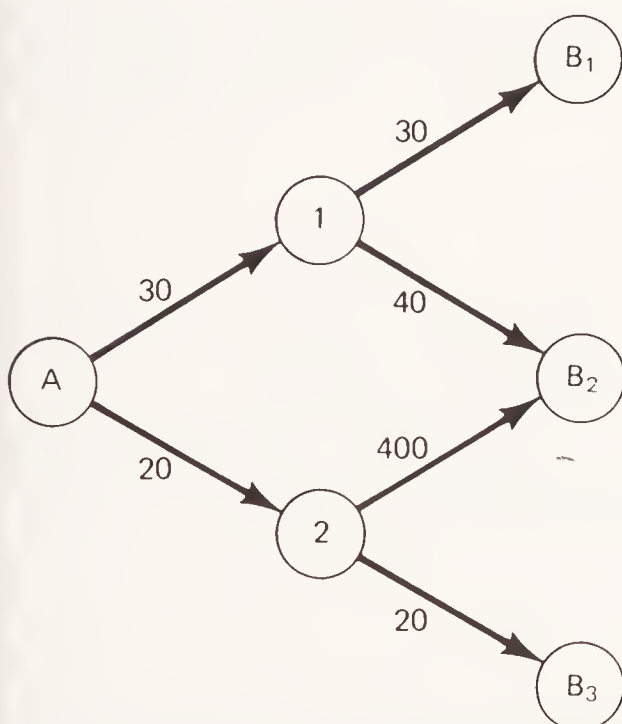


Figure 8-13 A transportation network

8.6. THE UNWARY TRAVELER

To make the problem in the proceeding section more interesting, let's insert a little uncertainty. Specifically, suppose our traveler is not yet familiar with the routes and occasionally makes mistakes. That is, although he chooses to go on the best route from a given node, he occasionally takes the other route by mistake. Let's suppose he does this 20 percent of the time.

First, let's consider the trip from the intermediate node 1 to the final destination. As before, the best call is up, since the cost is less than for the down route. But since our traveler will go down 20 percent of the time, the average cost is given by

$$0.8 \times 30 + 0.2 \times 40 = 32$$

This should be interpreted as an average cost of many trips when the traveler calls up on each trip.

Next, consider the trip from node 2 to the final destination. The best choice at this node is to call down. The average cost of many trips would be

$$0.8 \times 20 + 0.2 \times 400 = 96$$

In this case the high cost of the up route significantly affects the average cost.

From node *A*, the traveler may follow one of two routes to an intermediate node. If the traveler actually goes up from node *A* (regardless of whether he called up or down), the average cost of the trip would be the 30 units to go from *A* to node 1 plus the average cost of 32 units to go from node 1 to the final destination, or

$$30 + 32 = 62$$

for the complete trip. If the traveler actually goes down, the average cost would be 20 units to reach node 1 plus an average of 96 units to go from there to the final destination, or

$$20 + 96 = 116$$

for the complete trip. Therefore, the best call at node *A* would be up, since going this way would cost 62 units as compared with 116 for down.

But even if the traveler calls up at node *A*, he goes down 20 percent of the time. On the average, the cost of the entire trip would be

$$0.8 \times 62 + 0.2 \times 116 = 72.8$$

The best call to make at each of the nodes is illustrated in Figure 8-14. An interesting feature to note is that with the uncertainty in this problem, the

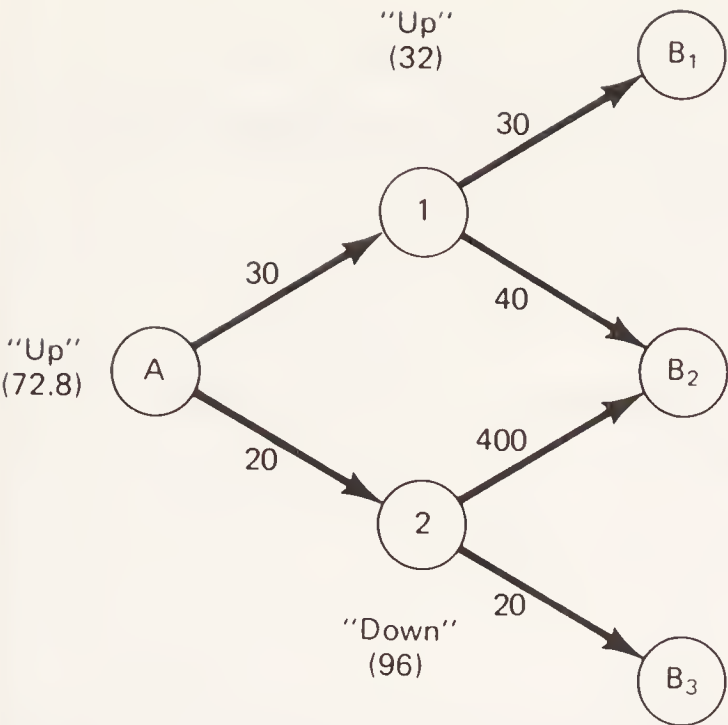


Figure 8-14 Transportation network showing expected costs

choice to be made at the intermediate node cannot be made while the traveler is at node *A*. That is, if he is at node 1, he should call up; if he is at node 2, he should call down. While he is at node *A*, he does not know with certainty which of the intermediate nodes he will reach, and therefore he cannot ascertain the proper call to make.

Another aspect of this problem is to restrain our traveler to make both calls while at node *A*. He has four possible choices:

- up—up
- up—down
- down—up
- down—down

To determine the average cost of each possible combination, we begin by computing the average cost of each possible call at each of the two intermediate nodes. These are

Node 1	up	$0.8 \times 30 + 0.2 \times 40 = 32$
Node 1	down	$0.8 \times 40 + 0.2 \times 30 = 38$
Node 2	up	$0.8 \times 400 + 0.2 \times 20 = 324$
Node 2	down	$0.8 \times 20 + 0.2 \times 400 = 96$

Now let's compute the average cost of the up-up and down-up calls. If the traveler goes up from node *A*, the average cost would be (he calls up at node 1)

$$30 + 32 = 62$$

If he goes down (he still calls up at node 2):

By calling up at node A, he goes up 80 percent of the time and down 20 percent; the average cost of the up-up call is given by

$$0.8 \times 62 + 0.2 \times 344 = 118.4$$

By calling down at node A, he goes up 20 percent of the time and down the remaining 80 percent, giving the average cost of

$$0.8 \times 344 + 0.2 \times 62 = 287.6$$

for the down-up call.

To compute the average cost of the up-down and the down-down calls, we begin by computing the average cost of actually going up or the cost of going down from node A, assuming the call of down is made no matter which intermediate node is reached:

going up from A:	$30 + 38 = 68$
going down from A:	$20 + 96 = 116$

The average costs of the up-down and down-down calls are

up-down	$0.8 \times 68 + 0.2 \times 116 = 77.6$
down-down	$0.8 \times 116 + 0.2 \times 68 = 106.4$

Inspecting the four results indicates that the best call is up-down at an average cost of 77.6.

When the traveler was permitted to delay making his choice at the intermediate node until he actually arrived at that node, the average cost was computed to be 72.8. This is 4.8 units less than when both calls must be made before leaving node A.

EXERCISES

8-1. The following is a simple game originally proposed by W. A. Wythoff. This game involves a set of objects that are divided into two heaps. Two players make alternate moves. At each move, the player has the option of removing any number of the objects from either of the heaps, or may remove an equal number of objects from both heaps. A player must take at least one object (that is, he cannot “pass”). He may remove all of the objects from one of the heaps, possibly removing a like number from the other heap. To repeat, the player may choose to diminish either heap, or both, to whatever extent he chooses, the only restriction being that if objects are removed from both heaps, an equal number

must be involved. The player who removes the last object, singly or with others, becomes the winner.

Suppose the game begins with two objects in one heap and three in the other. Assuming you make the first move, is there a move that will assure that you win regardless of how your opponent plays? Analyze this situation with the aid of a tree.

What if there are three objects in one heap and four in the other?

8-2. Another two-player game involving objects arranged in heaps is Grundy's game. Beginning, with a single heap of objects, the first player divides the original heap into two heaps, which must be unequal. The second player then divides one of these heaps into two heaps, which must also be unequal. This continues alternatively until it becomes impossible for one player to make a move. At this point, all heaps will contain either one or two objects. The last player to make a move wins.

For example, suppose the game begins with a heap of five objects. Player 1 divides them into heaps of one and four objects. Player 2 then divides the heap of four objects into heaps of one and three objects. Player 1 then divides the heap of three objects into heaps containing one and two objects. Player 2 now has no move and loses the game.

Suppose the game begins with a heap containing seven objects. Is there a move by which the first player is assured of winning? Analyze this game with the aid of a tree.

8-3. Suppose the game in the previous problem is changed so that the first player unable to make a move wins. That is, the objective is to always leave a move for your opponent. Now is there a strategy by which the first player to move can be assured of winning? Again use a tree in making the analysis.

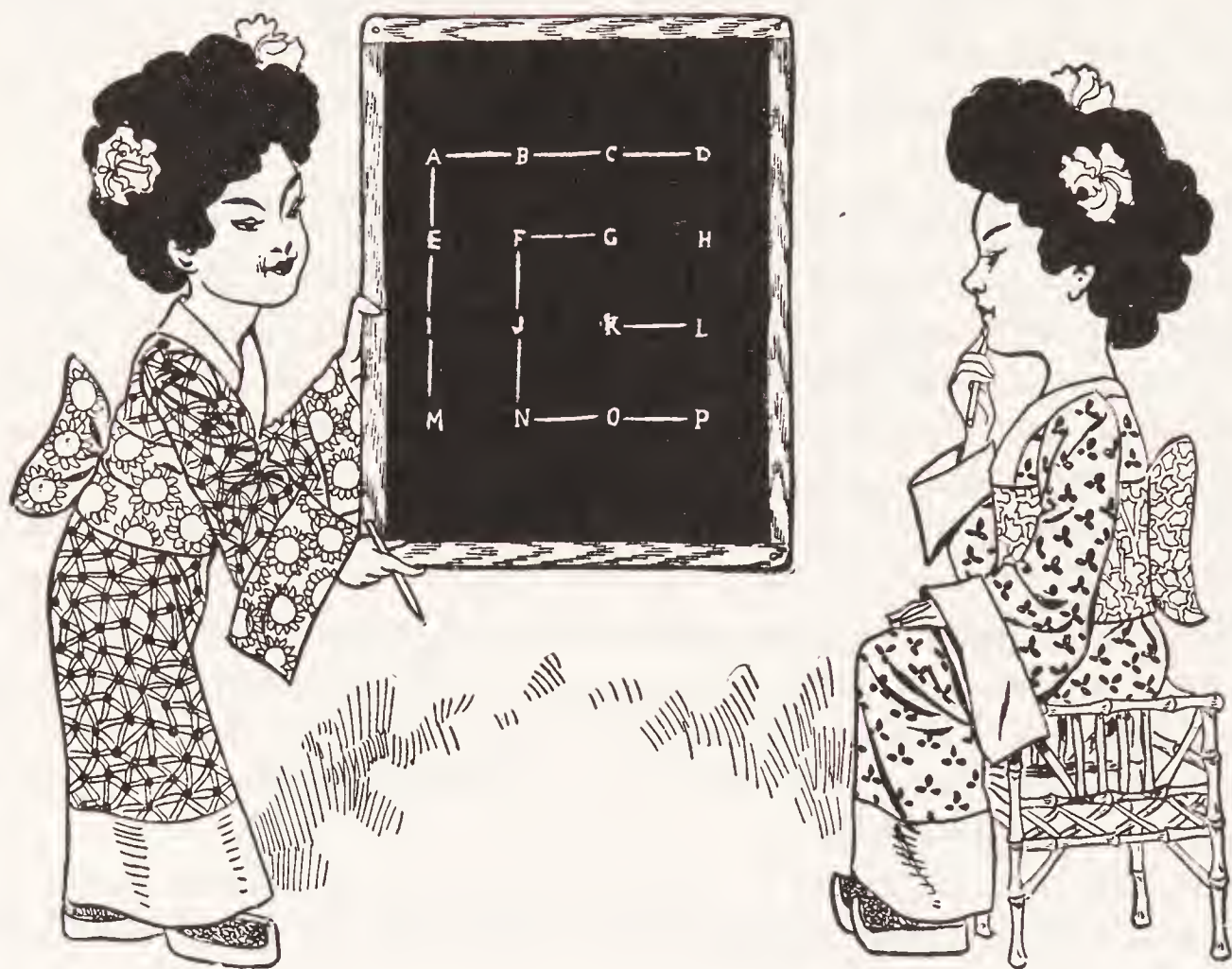
8-4. Below is a puzzle excerpted from *Mathematical Puzzles of Sam Loyd* edited by Martin Gardner (Dover Publications, New York, 1959, pp. 91-92):

Here is a familiar little puzzle game from the East, played upon lines very similar to the well known game of "Tit-Tat-Toe, three in a row." One of the Chinese girls writes sixteen letters on a slate in four rows, as shown. After marking a straight dash from A to B, she passes the slate to her opponent, who connects E with A. If the first player should now connect E with F, the other player would connect B with F and score "one box," and have the right to play again. But they have played so well that neither one has yet scored a box, although each has played six times. The game is reaching a critical point where one of them must win, for there are no draws in this game.

The little maiden sitting down has to play now, and if she connects M to N her opponent could score four boxes in one run, then having the right to one more play, would connect H and L which would win all the rest. What play would you now advise, and how many boxes will it win against the best possible play of the second player?

Remember, when a player scores a "box" he plays again. Suppose for example a player marks from D to H. Then the second player marks from H

THE BOXER'S PUZZLE



What is the best play and how many boxes will it win?

Reproduced by permission from *Mathematical Puzzles of Sam Loyd*, Martin Gardner, ed., Dover Publications Inc., New York, 1959.

to L, and no matter what mark the first player makes, the second player scores all nine boxes without stopping. It is a game that calls for considerable skill as you will discover after trying a few games.

Devise a tree that leads to the best possible move for the little maiden sitting down.

8-5. Suppose a man has twelve coins, which may all be of equal weight, or one coin may be heavier or lighter than the others. In three weighings on an equal-arm balance, it is possible to identify the odd coin, or to conclude that all coins are equal? Draw a tree that designates the proper procedure.

8-6. The following problem is taken directly from Henry E. Dudeney's *Amusements in Mathematics* (p. 22):

HEADS OR TAILS

Crooks, an inveterate gambler, at Goodwood recently said to a friend, "I'll bet you half the money in my pocket on the toss of a coin—heads I win, tails I lose." The coin was tossed and the money handed over. He repeated the offer again and again, each time betting half the money then in his

possession. We are not told how long the game went on, or how many times the coin was tossed, but this we know, that the number of times that Crooks lost was exactly equal to the number of times that he won. Now, did he gain or lose by this little venture?

The answer to this problem is completely independent of the amount of money Crooks has at the outset, so assume he has \$1.00.

8-7. Consider the following problem reported in *Scientific American* (April, 1972) and attributed to David L. Silverman. A quarter, a half-dollar, and a silver dollar are on a table. Smith takes one coin and Jones takes the other two. Upon tossing all coins, any coin falling tails is counted zero; any coin falling heads counts its value in cents. The tosser getting the larger score wins all three coins. If all three come up tails, the game is repeated. What coin should Smith choose so that the expected monetary win for each player is the same, that is, the game will be fair?

8-8. A man bought five barrels of wine and one barrel of beer. The wine was sold to two customers, one of whom bought twice as much as the other. In neither sale was the contents of any barrel manipulated. Which of the barrels in the accompanying figure contains the beer? Draw a tree that leads to the answer.



Reproduced by permission from *Amusements in Mathematics* by Henry E. Dudeney, Dover Publications Inc., New York, 1958.

8-9. Suppose you are given the three digits 3, 6, and 1. Draw a decision tree to determine if these digits can be arranged so as to form a number divisible by 8.

8-10. The toll on the Cross-Country Expressway is 90 cents. To make life simpler, the state legislature (after a 6-week debate in a special session) decided that the exact amount would be required of all motorists and that no pennies would be accepted. Draw a tree from which the maximum number of possible combinations of our coins (5, 10, 25, and 50 cents) totaling 90 cents can be determined. If desired, a program may be written.

8-11. The following puzzle is taken from *Mathematical Puzzles of Sam Loyd* edited by Martin Gardner (Dover Publications, New York, 1959, p. 54):

A friend and I were taking in the side shows at Coney Island the other day when we came to what the man told us was the squarest game on the beach. There were ten little dummies which you were to knock over with baseballs. The man said: "Take as many throws as you like at a cent apiece and stand as close as you please. Add up the numbers on all the men that you knock down and when the sum amounts to exactly 50, neither more nor less, you get a genuine 25¢ Maggie Cline cigar with a gold band around it."

Our money gave out before we learned how to win, and we noticed that lots of people didn't smoke any more Maggie Clines than we did. Can you show how we might have made exactly 50 points?

Develop the general structure for a tree to determine how many possible solutions exist to this problem (there is at least one), and then develop a program to determine the answer.



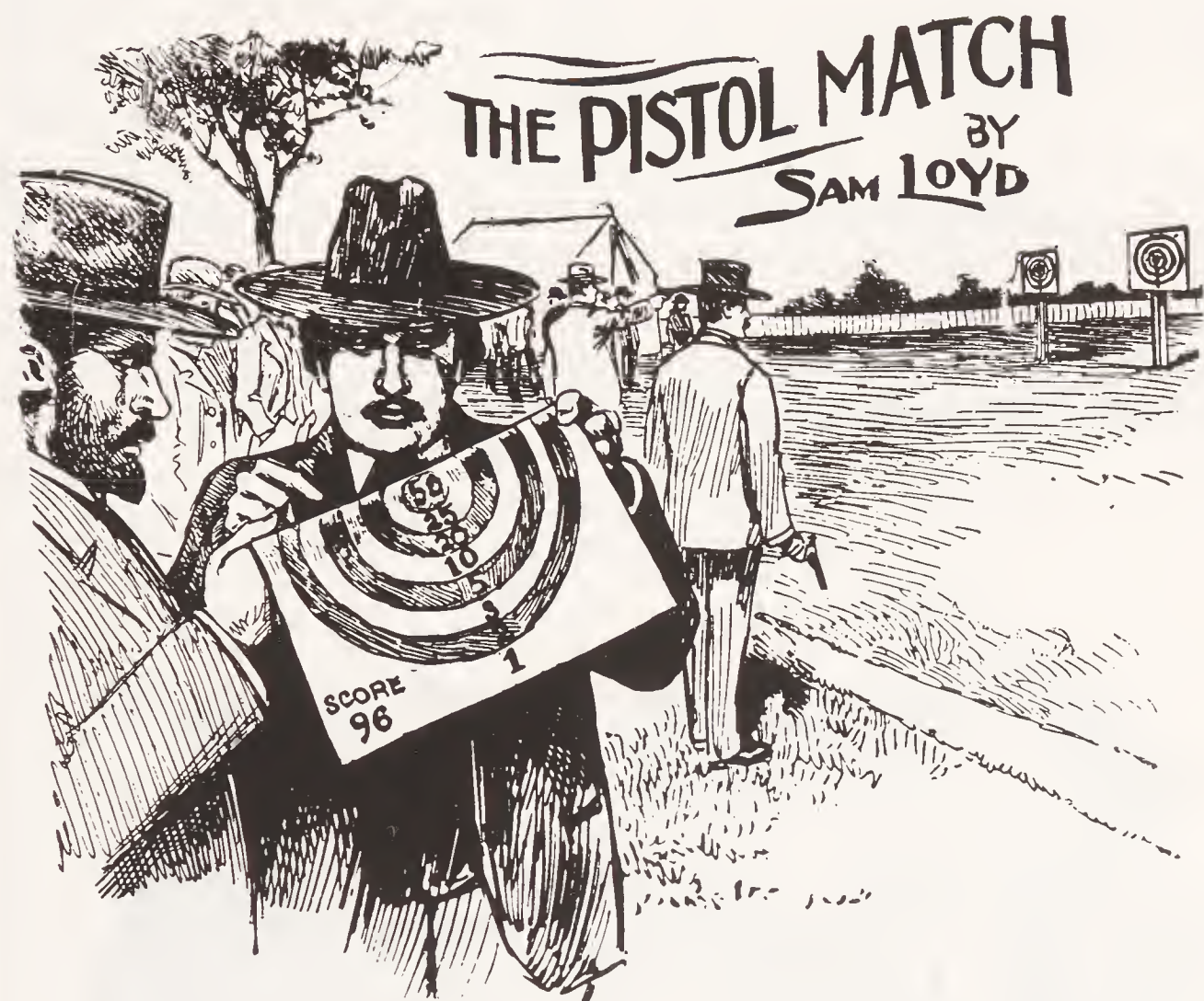
How can you score exactly fifty points?

Reproduced by permission from *Mathematical Puzzles of Sam Loyd*, Martin Gardner, ed., Dover Publications, Inc., New York, 1959.

8-12. The number 3,025 has the curious property that when it is separated into the two numbers 30 and 25 and then these two numbers added and the result squared, the final answer is 3,025. That is, $(30 + 25)^2 = (55)^2 = 3,025$. There is another number with four digits, all different, that exhibits this same property.

Devise the general structure for a tree that leads to the answer and then write a computer program to obtain the desired answer.

8-13. One of Sam Loyd’s puzzles involved a marksman who scored a 96 with six shots at the target illustrated in the accompanying figure. Furthermore, it was observed that the marksman had scored three “doublets,” a term used to refer to the feat of passing two bullets through the same hole. The objective was to find a way of making three doublets that give a score of 96. Draw a tree showing that there is one and only one solution to this problem. Alternatively, write a program to determine the answer.



Show how to score 96 with three “doublets.”

Reproduced by permission from *Mathematical Puzzles of Sam Loyd*, Martin Gardner, ed., Dover Publications Inc., New York, 1959.

8-14. There are three ways by which the nine digits (1 through 9) can be arranged to form three numbers with the properties that (1) the second number is twice the first and (2) the third number is three times the first. No digit may be used more than once. For example, the following three numbers satisfy these criteria:

- 192
- 384
- 576

292 Develop the general structure for a tree that leads to the other two combinations, and then prepare a program to obtain them.

8-15. The following amusing problem appeared in Henry E. Dudeney’s *Amusements in Mathematics* (Dover Publications, New York, 1958, p. 14). The nine digits (excluding zero) are arranged so as to comprise one three-digit number and three two-digit numbers. The product of the three-digit number and one of the two-digit numbers is computed, and the product of the two remaining two-digit numbers is computed, as illustrated by the following example:

158

23

3,634

79

46

3,634

The digits are to be arranged so that the two products are equal. The objective is to determine the arrangement of digits that gives the largest possible product. Products of 5,568, 7,008, and 7,448 have been found.

One approach to this problem is to perform an exhaustive search. We might begin by reserving nine locations for each of the digits, that is,

$X_1 X_2 X_3$

A

$X_4 X_5$

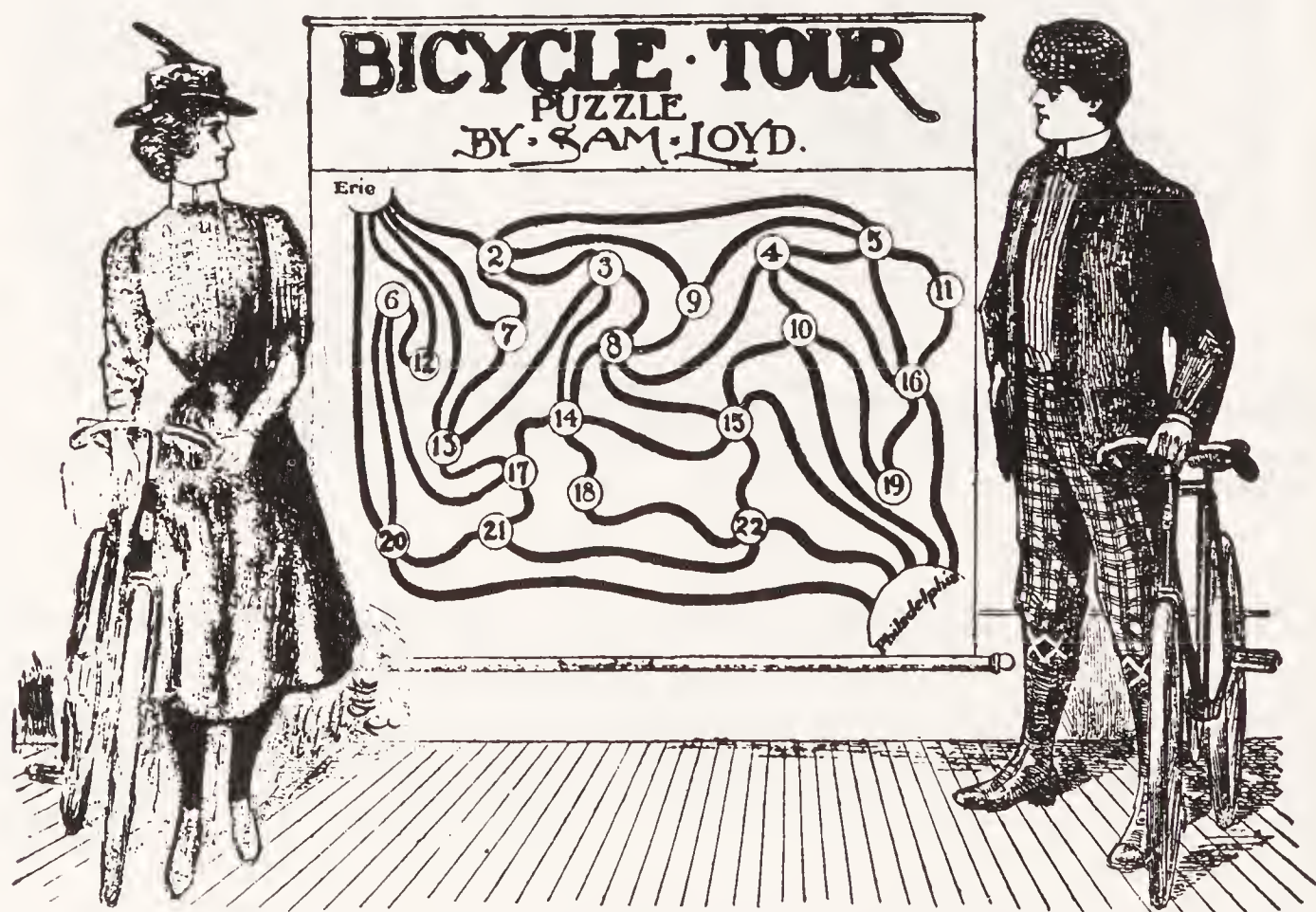
B

$X_6 X_7$

C

$X_8 X_9$

D



Reproduced by permission from *Mathematical Puzzles of Sam Loyd*, Martin Gardner, ed., Dover Publications Inc., New York, 1959.

The first three digits comprise number A, the next two number B, etc. Then the product of A and B can be compared with the product of C and D. If equal, the result can be compared with the largest product found so far.

The problem is to devise a tree structure that if followed will cause all possible combinations of the nine digits to appear in array X. This can then be implemented in a computer program to obtain the desired answer, but be prepared for a long run.

8-16. The map shown in the figure on page 292 is from a puzzle that originated with Samuel Loyd. The objective of the game was to determine the route by which the cyclist could go from Erie to Philadelphia while passing through every town once but not traversing any road twice. Suggest an appropriate mechanism for representing the grid in the computer's memory.

8-17. The accompanying figure shows a map taken from a puzzle by Sam Loyd which represents the "newly discovered cities and waterways on our nearest neighbor planet, Mars." The objective was to start at the city marked *T* at the south pole and spell a complete English sentence by making a tour of all the cities, visiting each city only once, and returning to the starting point.

Prepare a means by which this map can be represented in the computer's memory.



The Canals on Mars

Reproduced by permission from *Mathematical Puzzles of Sam Loyd* ed. by Martin Gardner, Dover Publications Inc., New York, 1959.

9

COMPILATION

Chapter 6 introduced the basic instructions available on a typical general-purpose digital computer. At that time, it was noted that very rarely is programming ever done at this level. Current practice is to write programs in a language such as Fortran, PL/I, Basic, or their counterparts. Statements in these languages bear little similarity if any to the basic machine instructions generally available.

The process of translating statements in a problem-oriented language into machine language instructions is referred to as compiling. In this chapter we shall consider several aspects of the compilation process.

9.1. TREE REPRESENTATION OF EXPRESSIONS

In Chapter 2 we introduced the concept of an expression and a hierarchy for the mathematical operations involved. For example, consider the following quadratic expression:

$$A \times X \uparrow 2 + B \times X + C$$

To evaluate this expression from the available values of A, B, C, and X, the operations should be performed in the following order (quantities in brackets are computed in earlier steps):

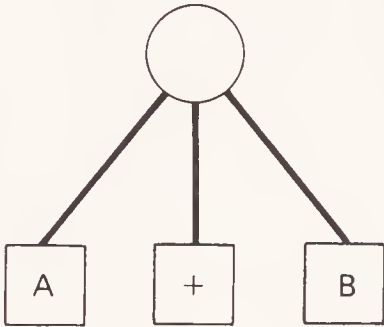
1. Compute $X \uparrow 2$
2. Compute $A \times [X \uparrow 2]$
3. Compute $B \times X$
4. Compute $[A \times X \uparrow 2] + [B \times X]$
5. Compute $[A \times X \uparrow 2 + B \times X] + C$

The sequence of the operations depends upon the hierarchy, not upon the order in which they appear in the expression.

Let's examine how one can analyze an expression to determine the proper sequence for performing mathematical operations. One procedure is to represent the expression in the form of a tree. As a convention, we will represent a mathematical operation by a node with three branches, two for the quantities involved and one for the operator (which will always be the center branch). For example, the expression

$$A + B$$

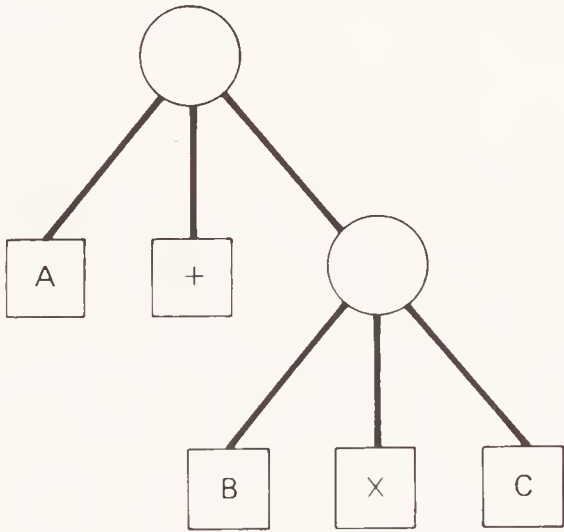
will be represented as follows:



Either of the variables may be replaced by another node representing a subexpression. For example, the expression

$$A + B \times C$$

is represented as follows:



By extending this procedure, very complex expressions can be represented. The representation of the expression

$$A \times X \uparrow 2 + B \times X + C$$

is given in Figure 9-1.

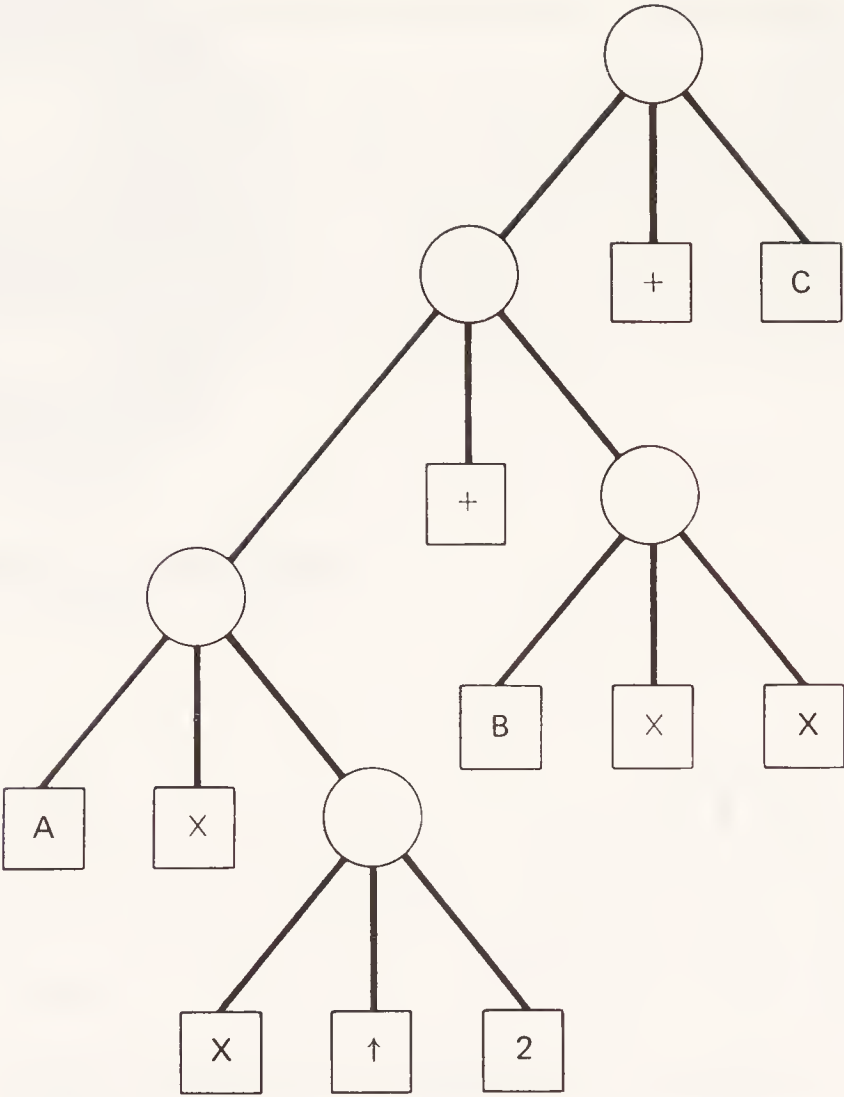


Figure 9-1 Tree representation of the expression $A \times X \uparrow 2 + B \times X + C$

9.2. THE POLISH NOTATION

An expression such as

$$A \times X \uparrow 2 + B \times X + C$$

is said to be in the *infix notation* since the mathematical operators separate the two quantities upon which the respective operation is to be performed. Another characteristic of the customary infix notation is that the proper sequence in which the operations are to be performed is generally not the order in which they appear in the expression. Instead, this order is to be according to the customary hierarchy. In evaluating such an expression directly, several scans across the expression must be made in order to perform the operations in their proper sequence.

In the postfix notation, the mathematical operator follows the two quantities upon which the operation is to be performed. For example, the infix expression $A \times B$ would be represented as $AB \times$ in the postfix notation. At this point, we encounter a slight notational problem. If we permit variable names to assume more than one character, postfix expressions become ambiguous. For example, the postfix expression $ABC \times$ could be interpreted as either $A \times BC$ or $AB \times C$ in infix notation. In practical compilers, special delimiter symbols are placed

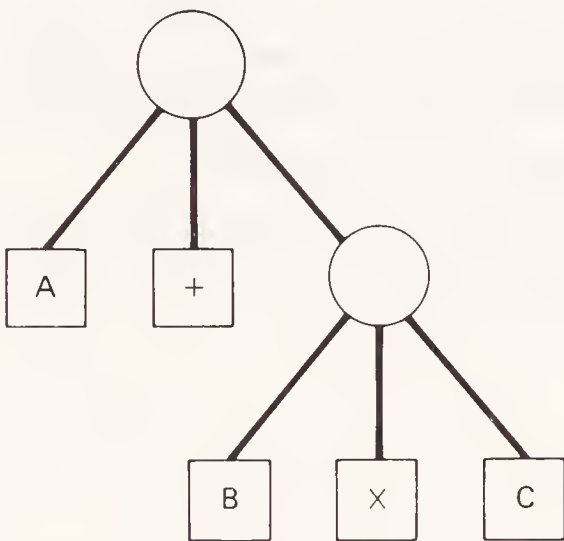
before or after all variable names, but in this chapter we shall take the simpler approach of restricting all variable names to one alphabetic character.

The Polish notation is a form of the postfix expression in which the mathematical operators appear in the order in which they are to be performed. From this notation, the expression can be evaluated by a single scan from left to right, performing the mathematical operations as they are encountered. Expressions written in this form are referred to as *Polish strings*.

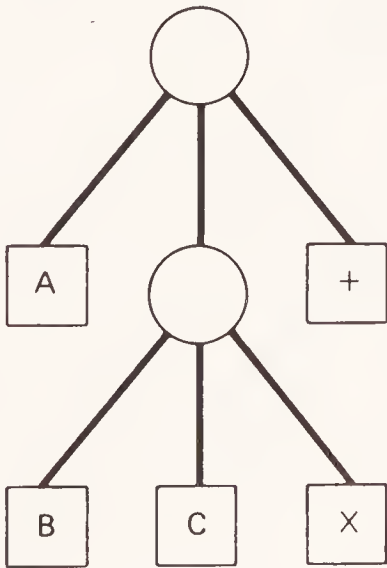
As our first example of a Polish string, consider translating the infix expression

$$A + B \times C$$

As we have already seen, this expression can be represented by the following tree:



To obtain the representation in postfix form, we need only to interchange the right two branches at each node, obtaining



The Polish string is obtained by writing down the postfix expression beginning at the bottom of the tree and working upwards. That is, we first write down BCX for the lower node. Doing the same for the upper node gives the final postfix expression

$$ABC \times +$$

In evaluating the expression, we scan from left to right. Upon encountering the first operator, the $\times$, we multiply B and C to give a single value, which in effect replaces the characters BCX in the Polish string. Using brackets to represent previously evaluated quantities, the Polish string becomes

$$A[BC \times] +$$

Proceeding with the scan from left to right, the next mathematical operator encountered is the $+$, which means to add the two previous values in the expression, namely, A and $[BC \times]$.

As a slightly more complicated example, we might determine the Polish string representation for the infix expression

$$A \times X \uparrow 2 + B \times X + C$$

whose tree representation was given in Figure 9-1. The next step is to interchange the two rightmost branches at each node, giving the result in Figure 9-2. Since we shall permit only single-character variables in our Polish notation, the constant 2 has been replaced by the variable N .

We can now start with the lowest subexpression on the leftmost branch and work upwards. First, we obtain

$$XN \uparrow$$

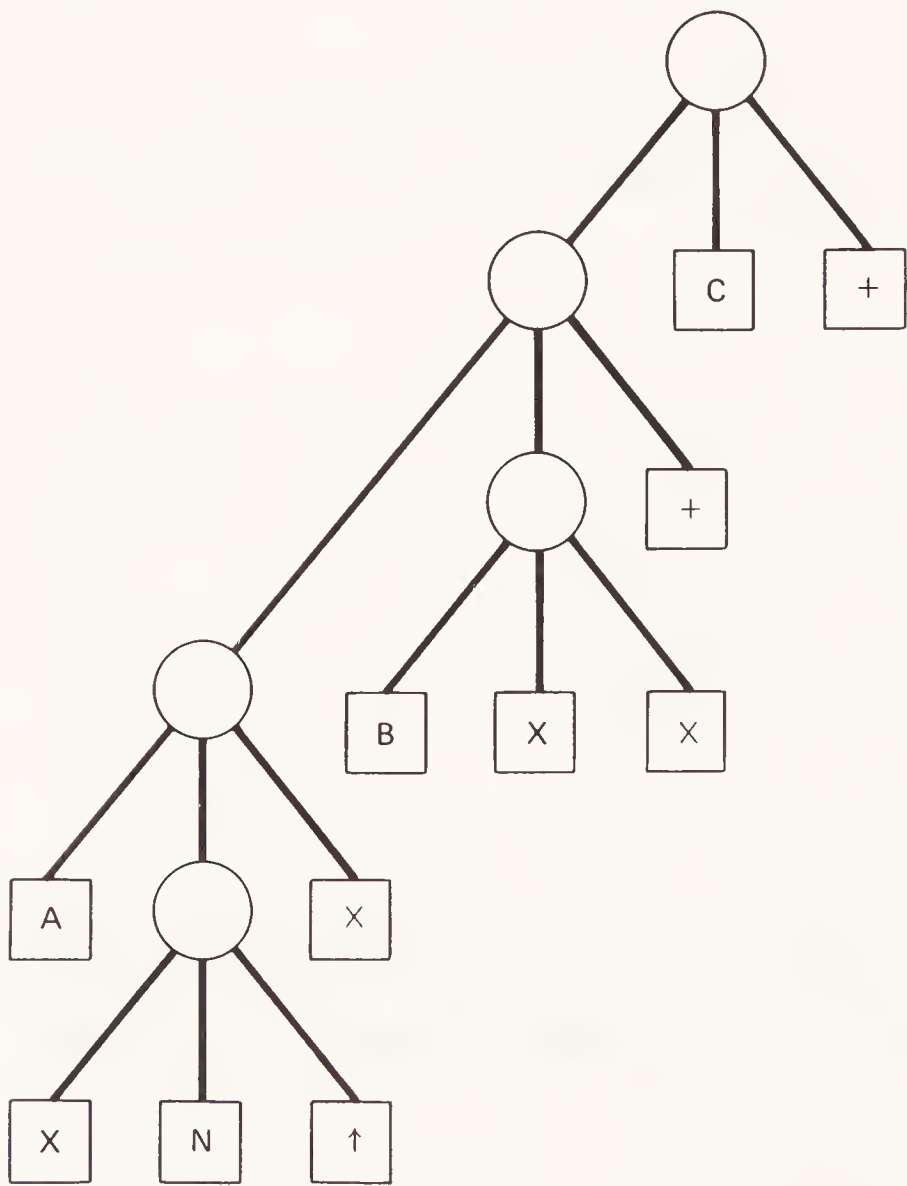


Figure 9-2 Interchanging the two rightmost branches of the tree in Figure 9-1

Then proceeding to the next level, we obtain:

$$AXN \uparrow X$$

At the next node upward, we encounter another node on the end of the second branch. Therefore, we go to the bottom of this arm and work upward, forming another Polish expression. At the lowest node, we have the expression

$$BX \times$$

We now have Polish expressions for the two arms of the second node (from top), which can be combined along with the operator to give

$$A \times N \uparrow X \times BX \times +$$

Incorporating the top node gives the postfix expression

$$AXN \uparrow X \times BX \times + C +$$

This is the Polish string equivalent to the infix expression

$$A \times X \uparrow N + B \times X + C$$

To illustrate the evaluation of Polish strings, suppose we compute the numerical value of the previous expression for the following values of the variables:

$$\begin{array}{ll} A = 2 & X = 4 \\ B = -1 & N = 2 \\ C = 3 \end{array}$$

The steps in evaluating the postfix expression are illustrated in Figure 9-3. Note that the operators are computed in the same order as they appear in the Polish string.

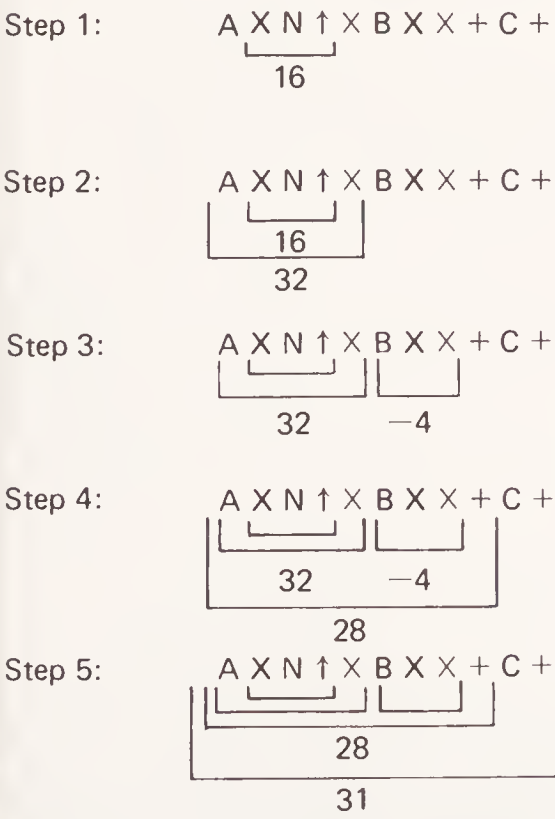
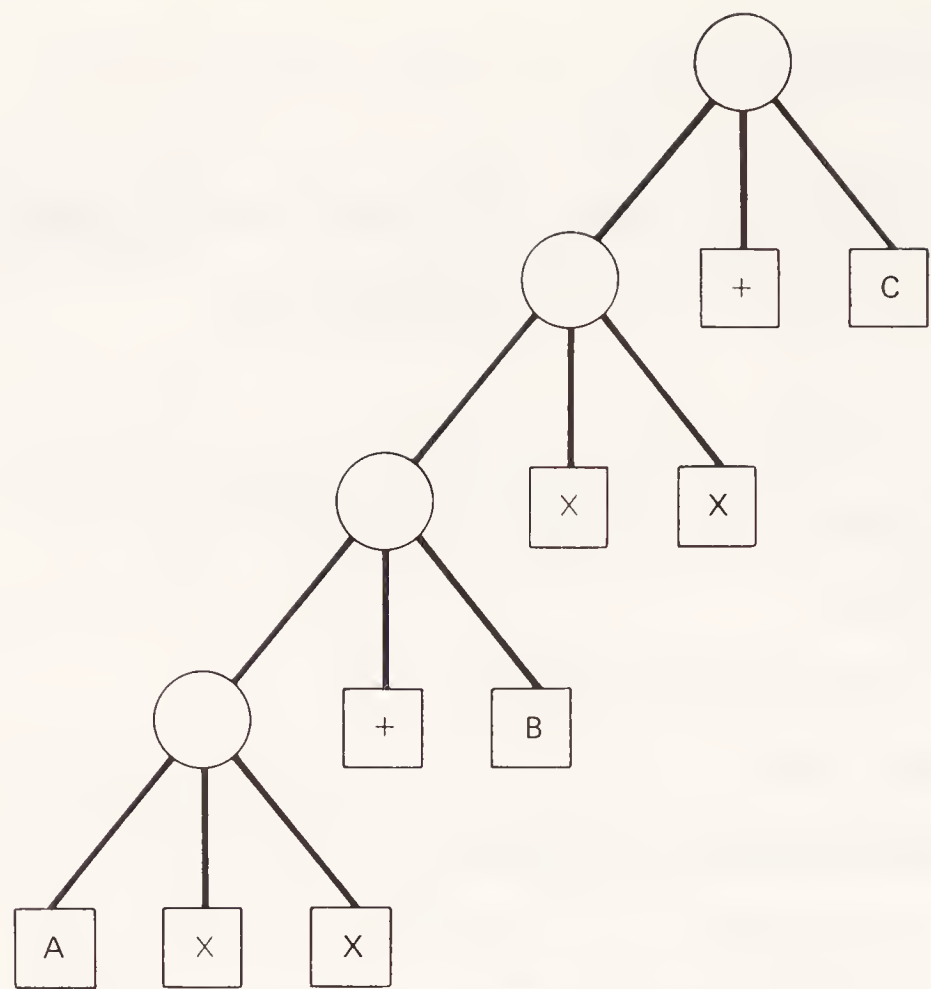
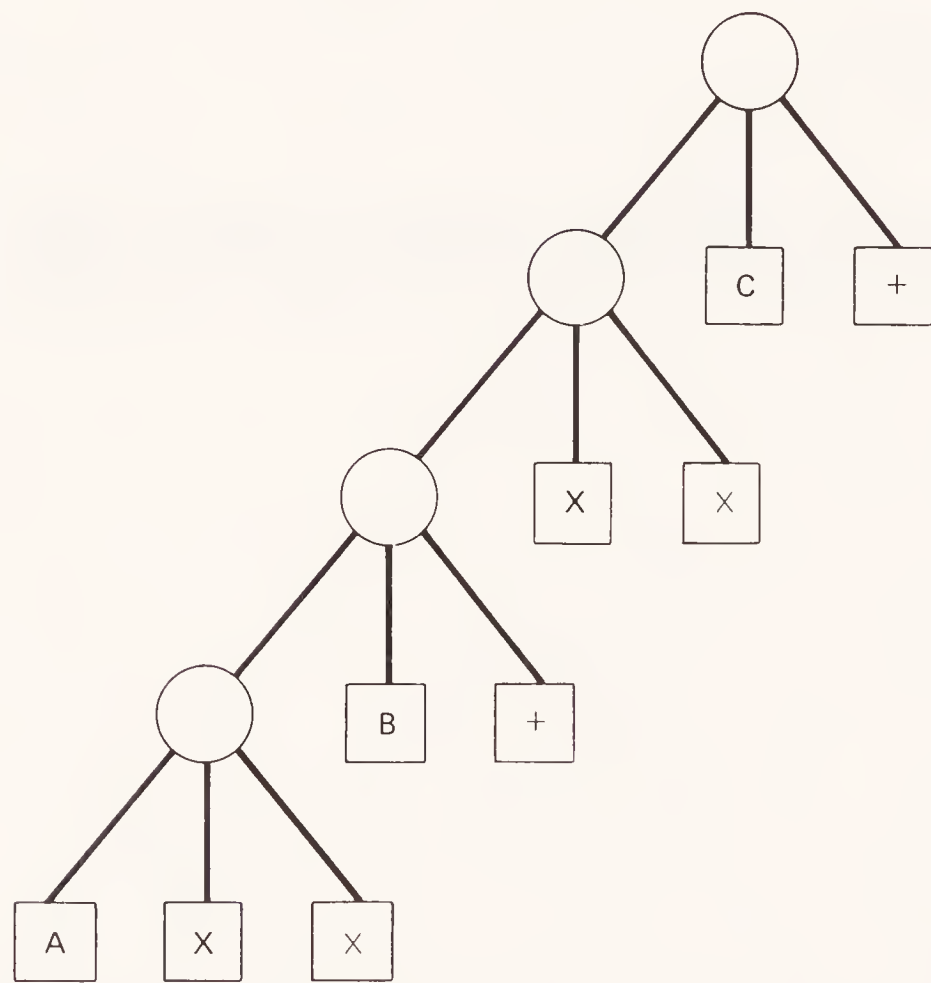


Figure 9-3 Evaluation of the Polish expression



a. Tree representation of the infix expression



b. Tree representation of the postfix expression

Figure 9-4 Tree representations of the expression $(A \times X + B) \times X + C$.

Another interesting characteristic of the Polish notation is that it never contains parentheses, even though the infix notation of the expression might contain several pairs. For example, consider the following alternate form of the expression for a quadratic polynomial:

$$(A \times X + B) \times X + C$$

To transform this expression into a Polish string, we again construct the tree representation (Figure 9-4a) of the infix expression and then interchange the rightmost two branches at each node (Figure 9-4b). Starting at the bottom of the tree in Figure 9-4b and working upward gives the Polish string

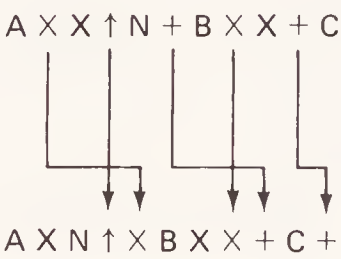
$$A \times X \times B + X \times C +$$

which contains no parentheses.

9.3. AN ALGORITHM FOR THE TRANSLATION PROCESS

Developing the Polish string from the infix notation via tree representation suffers from not being readily expressed in algorithmic language. In this section we shall present an algorithm by which an expression can be translated from infix to postfix in a single scan.

To develop our algorithm we shall begin by examining an infix expression and its postfix equivalent:



Note the following characteristics:

- 1. The order of the variables is the same in both the infix and the postfix notation.
- 2. When successive operators in the infix notation are such that the first operator is lower in the hierarchy than the second operator, their order of appearance in the postfix expression is reversed.

These two features can be incorporated into an algorithm with the aid of an auxiliary list that we shall call the stack, which operates on the principle of first in, last out.

The algorithm proceeds with a scan of the infix expression from left to right. The steps are as follows:

- 1. If the character examined is a variable, place it directly into the postfix expression.
- 2. If the character examined is an operator, then
 - a. Move operators from the stack to the postfix expression until an operator is reached whose level of hierarchy is lower than that of the current operator, and
 - b. place the current operator in the stack.

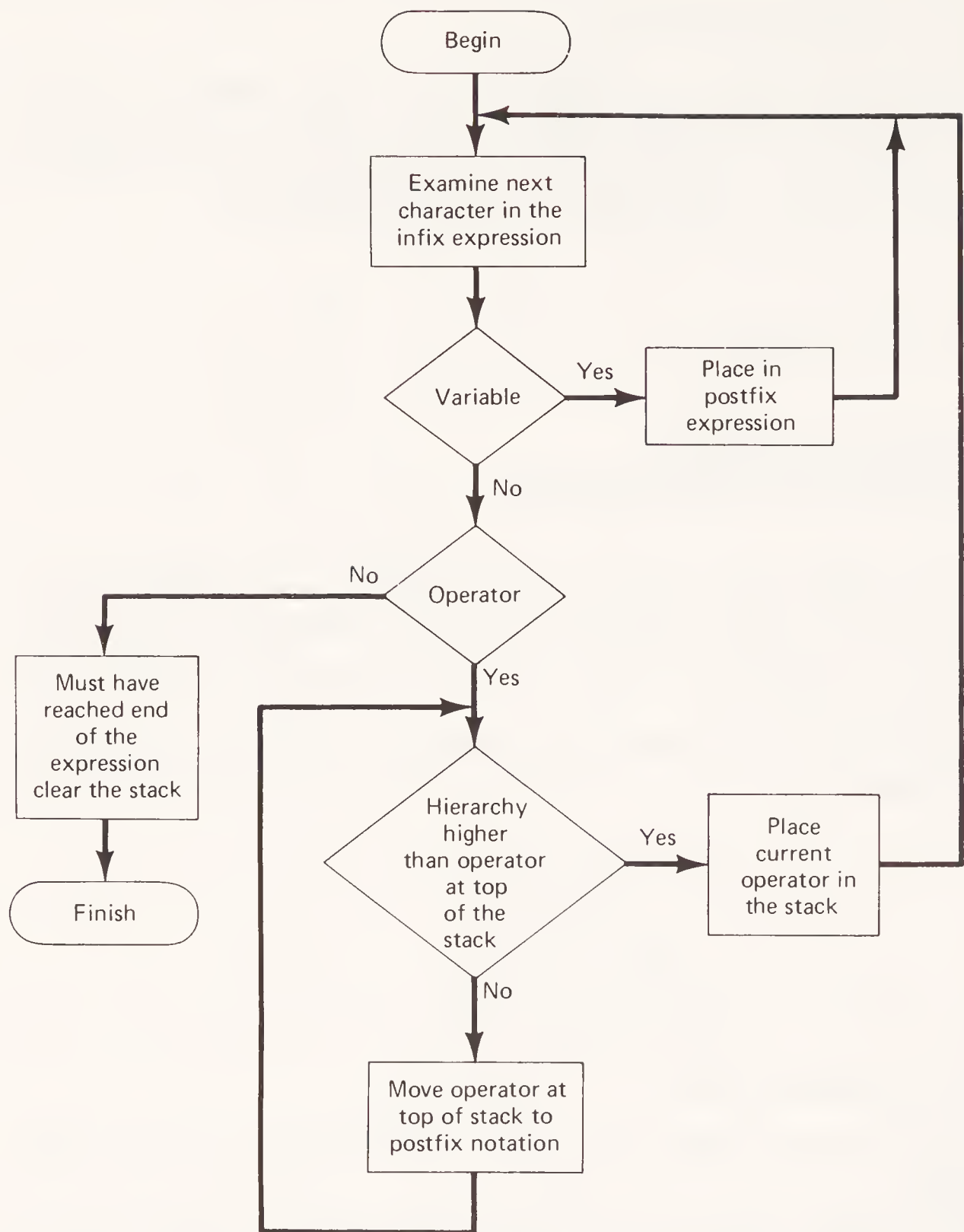


Figure 9-5 Flowchart of algorithm for translating from infix to postfix

3. Upon reaching the end of the infix expression, clear the stack.

This procedure is represented in flowchart form in Figure 9-5.

An illustrative example of this process is given in Figure 9-6 in which the quadratic expression

$$A \times X \uparrow N + B \times X + C$$

is converted from infix to postfix. The steps are as follows:

1. *Character A*: Variable; place directly into postfix expression.
2. *Character X*: Operator; place into stack, which was empty.
3. *Character X*: Variable; place directly into postfix expression.
4. *Character ↑*: Operator; hierarchy is higher than operator (X) at top of stack; place into stack.

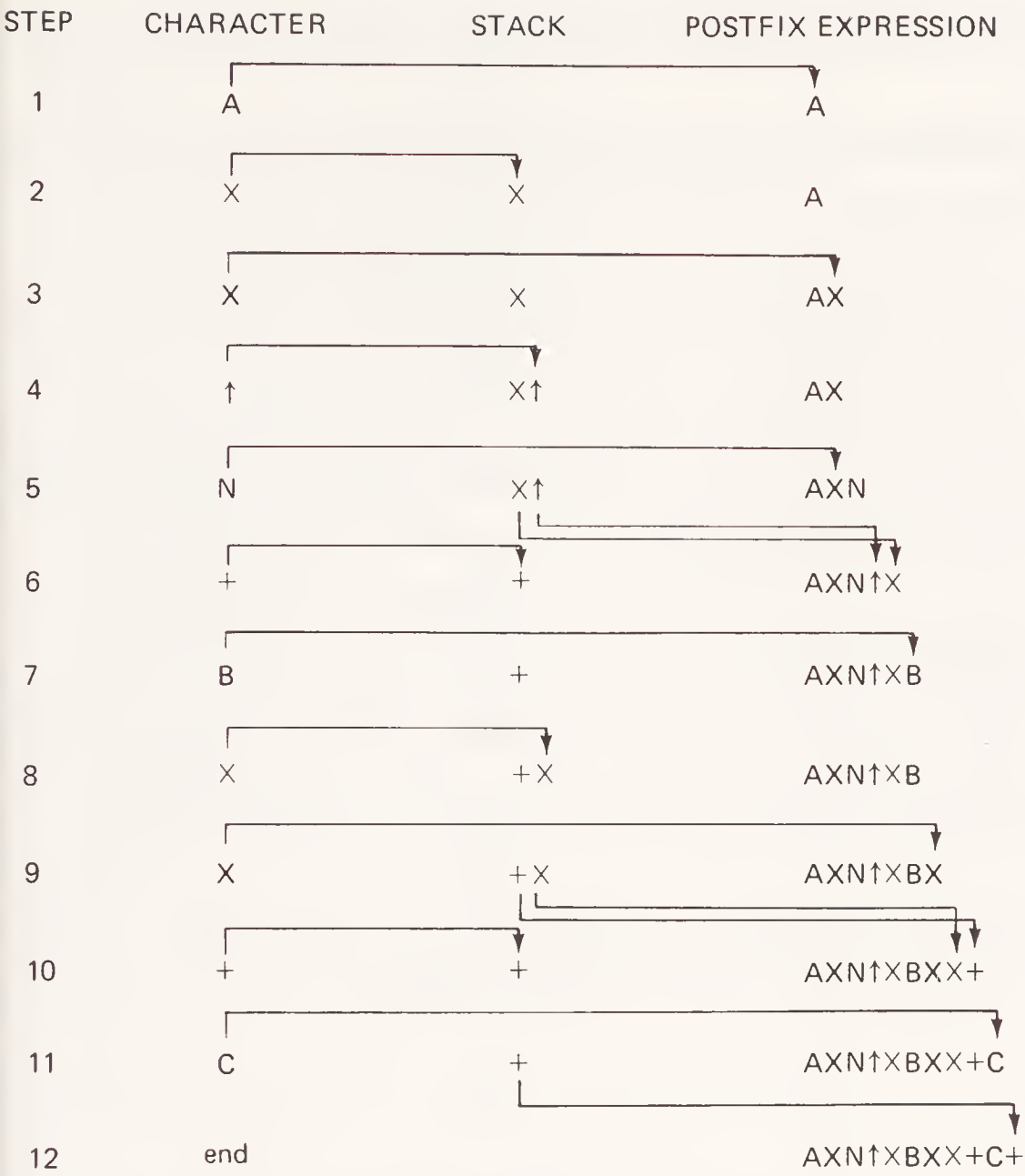


Figure 9-6 Example of the translation process

- 5. Character N: Variable; place directly into postfix expression.
- 6. Character +: Operator; move ↑ from stack to postfix expression, move X from stack to postfix expression; place + into stack.
- 7. Character B: Variable; place directly into postfix expression.
- 8. Character X: Operator; hierarchy is higher than operator (+) at top of stack; place into stack.
- 9. Character X: Variable; place directly into postfix expression.
- 10. Character +: Operator; move X from stack to postfix expression; move + from stack to postfix expression; place + into stack.
- 11. Character C: Variable; place directly into postfix expression.
- 12. End of infix expression: Move + from stack to postfix expression.

This procedure can be readily extended to encompass expressions containing parentheses. Open parentheses are simply placed directly into the stack. Upon encounter of a close parenthesis, operators are removed from the stack until an open parenthesis is encountered, and the pair of parentheses is discarded. Incorporating this into the flowcart in Figure 9-5 gives the flowchart in Figure 9-7.

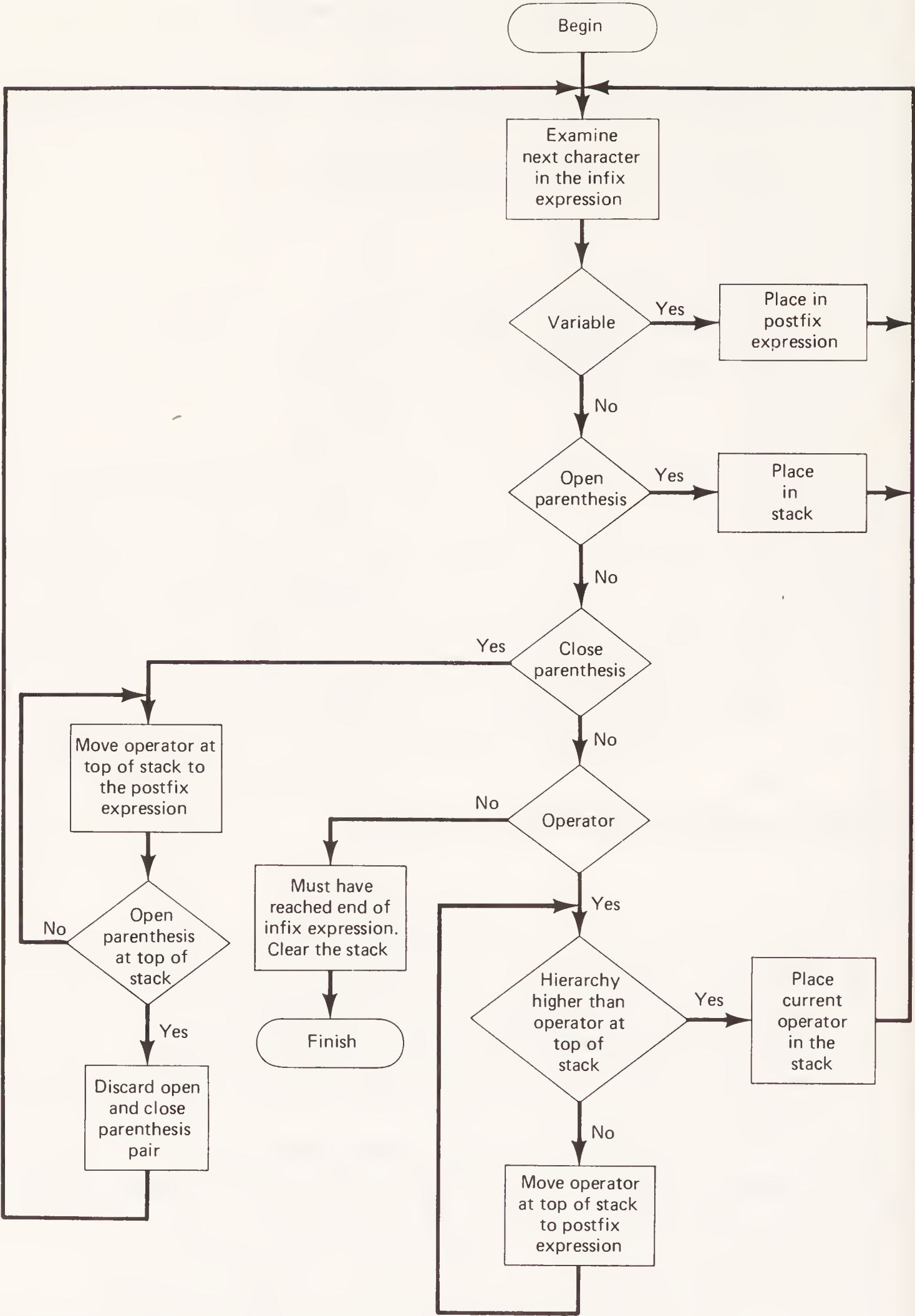


Figure 9-7 Flowchart of algorithm for translating from infix to postfix

As an example, suppose we translate the expression

$$(A \times X + B) \times X + C$$

into the postfix notation. The steps are given in Figure 9-8 and are explained as follows:

- 1. *Character (*: Open parenthesis; place directly in stack.
- 2. *Character A*: Variable; place directly in postfix expression.
- 3. *Character X*: Operator: place directly into the stack.
- 4. *Character X*: Variable; place directly in postfix expression.
- 5. *Character +*: Operator; hierarchy is lower than operator (X) at top of stack; move X to postfix expression; place + in stack.
- 6. *Character B*: Variable; place directly in postfix expression.
- 7. *Character)*: Close parenthesis; move + from stack to postfix expression; discard open parenthesis in stack; discard close parenthesis.
- 8. *Character X*: Operator; place in stack.
- 9. *Character X*: Variable; place directly in postfix expression.
- 10. *Character +*: Operator; hierarchy is lower than operator (X) at top of stack; move X from stack to postfix expression; place + in stack.
- 11. *Character C*: Variable; place directly in postfix expression.
- 12. *End of infix expression*: Move + from stack to expression.

STEP	CHARACTER	STACK	POSTFIX EXPRESSION
1	(	(	
2	A	(	A
3	X	(X	A
4	X	(X	AX
5	+	(+	AXX
6	B	(+	AXXB
7	)		AXXB+
8	X	X	AXXB+
9	X	X	AXXB+X
10	+	+	AXXB+XX
11	C	+	AXXB+XXC
12	end		AXXB+XXC+

Figure 9-8 Translation of the infix expression (A X X + B) X X + C to postfix

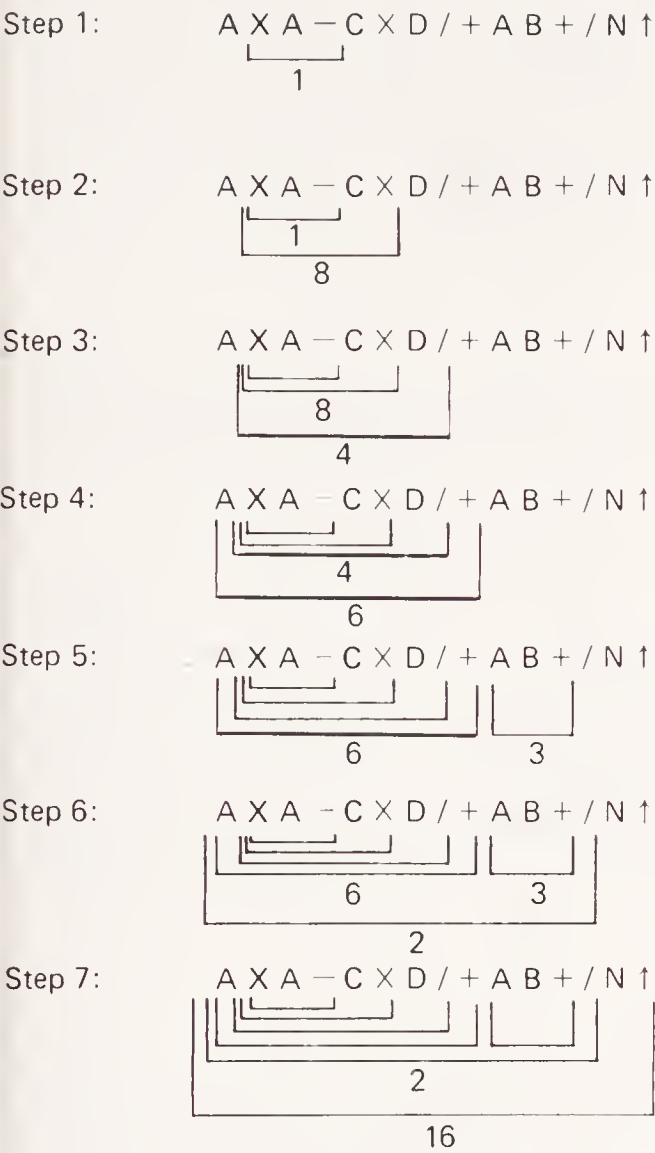


Figure 9-10 Evaluation of a postfix expression

9.4. ARITHMETIC ASSIGNMENT STATEMENTS

As pointed out in Chapter 2, an arithmetic assignment statement essentially consists of an expression and the “replacement operator,” which we have represented by the reverse arrow. Using the postfix notation, the replacement operator can be treated much like the other arithmetic operators. That is, the statement

$$A \leftarrow B$$

is interpreted as “store the current value of B in the location reserved for A.” This can also be represented by the postfix notation

$$AB \leftarrow$$

In translating to the Polish notation, the entire arithmetic assignment statement can be converted if we will only treat the replacement operator $\leftarrow$ as being lower on the hierarchy than any of the usual arithmetic operators.

As an example, the conversion of the arithmetic assignment statement

$$A \leftarrow (B + C) / (A - J \times K)$$

to Polish notation is illustrated in Figure 9-11.

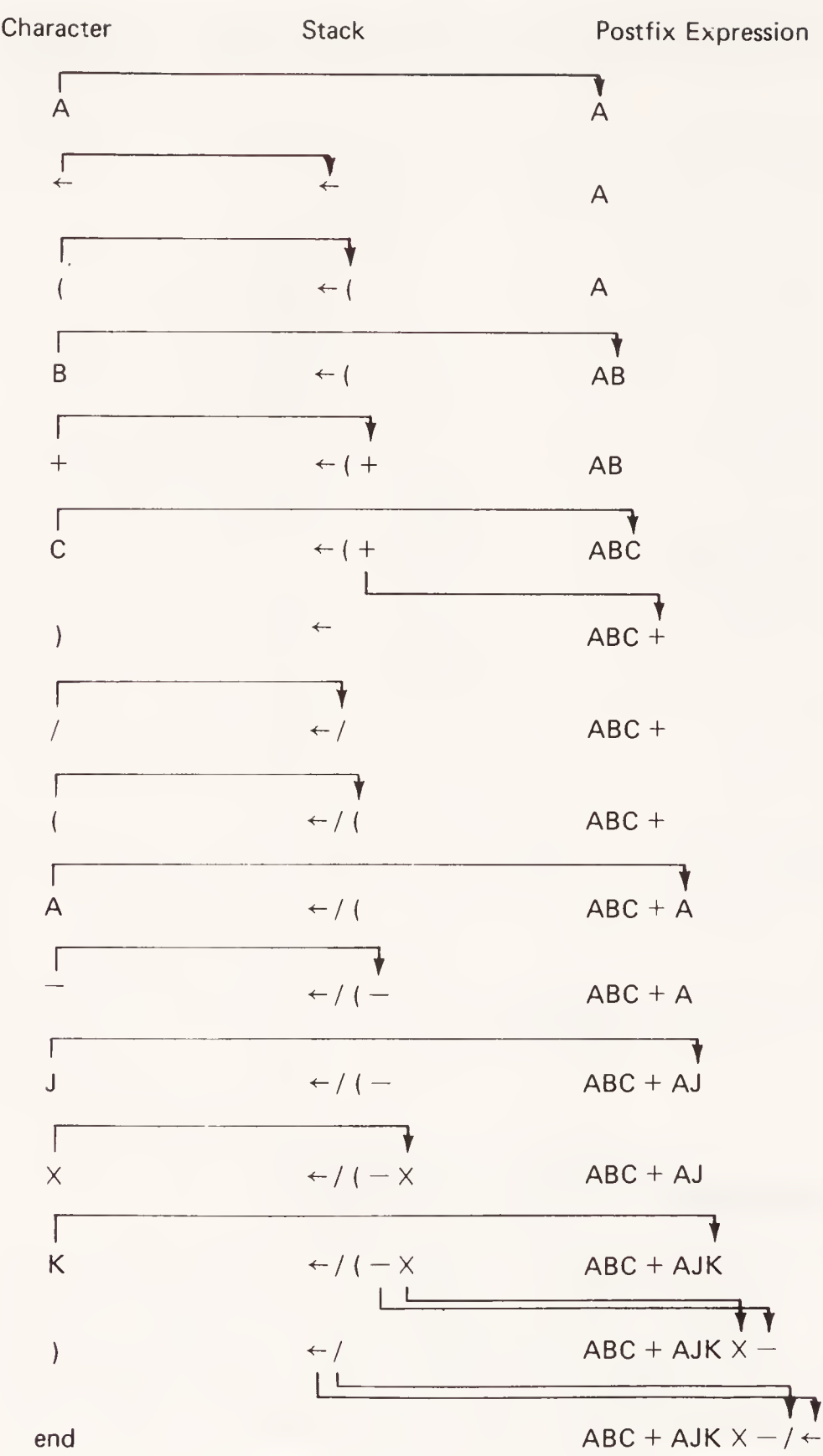


Figure 9-11 Translation of an arithmetic assignment statement

9.5. TRANSLATING POSTFIX EXPRESSIONS TO ASSEMBLY CODE

The ultimate objective of the compilation process is to generate the machine-executable code that is equivalent to the source statement being compiled. The basic objective of the translation from infix to postfix is to arrange the operators in the order in which they should be executed. The next step is to generate the assembly-language instructions to perform these operations.

The five arithmetic operators +, -, X, /, and ↑ are certainly executable by a

digital computer, but at the machine-language level are treated somewhat differently. The operators $+$ and $-$ are generally the most straightforward. For simplicity, let's assume that we are working with a computer that has only one arithmetic register, namely, the accumulator. In this case, addition is generally obtained with an instruction such as

ADD G

which is interpreted as "add G to the current contents of the accumulator." For subtraction, an instruction such as

SUB G

is used.

In most computers, multiplication of two numbers generally results in a doubleword product, which may require special handling. Similarly, division often requires that the dividend be extended to a doubleword before the operation is commenced, which also requires some special handling. For simplicity, however, let's assume that we have instructions such as

MPY G
DIV G

which operate through the accumulator in the same fashion as ADD and SUB.

On most machines, the exponentiation operation is generally accomplished via a function rather than by a specific hardware instruction. We shall, however, again assume that this operation is accomplished by the instruction PWR (short for "power") that also operates through the accumulator.

In addition to these five "instructions," we shall also assume that our machine has the "load accumulator" instruction, for example,

LDA G

and the "store accumulator" instruction, for example,

STO G

Of course, the replacement operator, $\leftarrow$, is essentially equivalent to the STO instruction.

The process of translation to assembly language can essentially be accomplished with a single scan of the postfix expression. To assist in this process, we will again need to use a "stack" for temporary storage. In essence, this stack will function in exactly the same manner as the stack that was used in translating from infix to postfix. That is, it operates on the last-in, first-out principle.

In this case, however, we need a notation for elements in the stack. We shall use the notation

STK-0

to represent the entry at the top of the stack; the notation STK-1 to represent the next-to-top entry, STK-2 to represent second from top, etc.

Furthermore, we shall need some temporary storage locations to store inter-

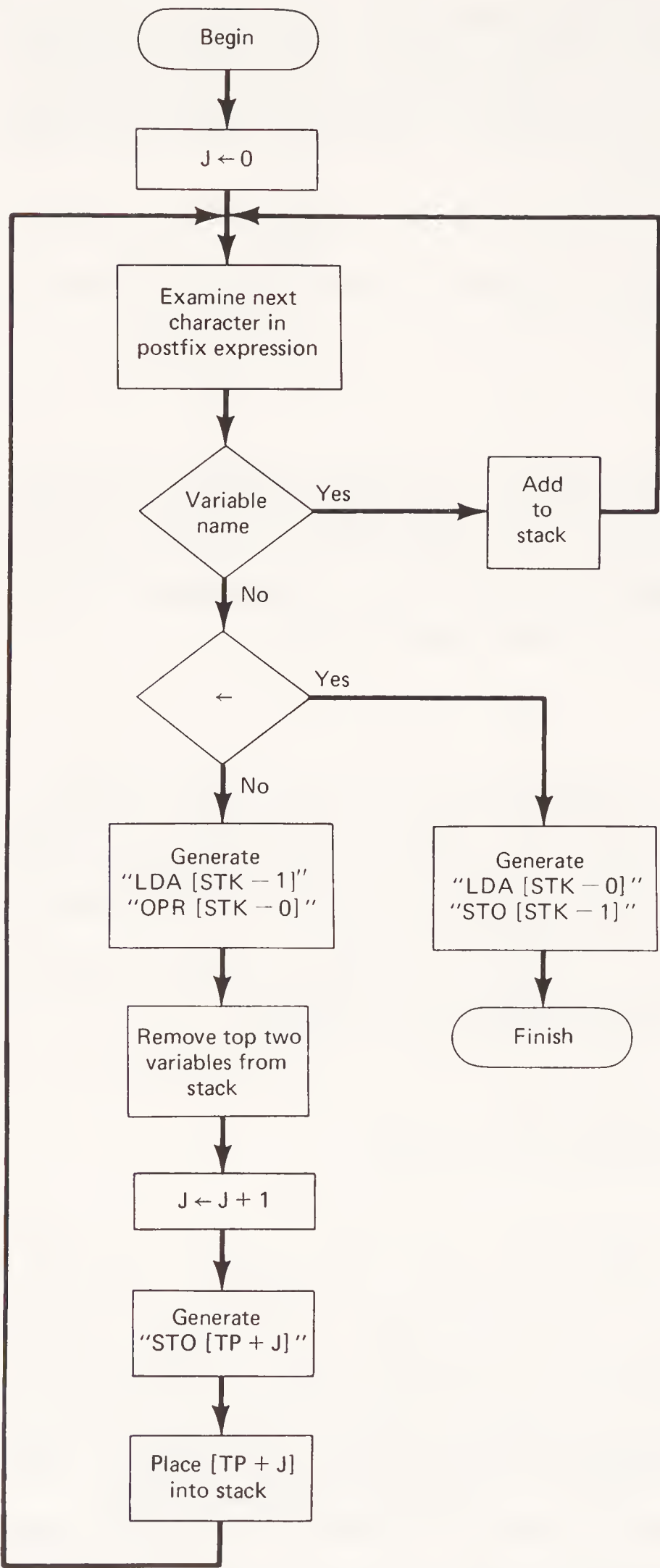


Figure 9-12 Algorithm for translating from postfix to assembly language

mediate results encountered during evaluation of the expression. We shall represent the address of the first such element as $[TP + 1]$, the second as $[TP + 2]$, the j as $[TP + J]$, etc.

Figure 9-12 gives the flowchart of a simple algorithm for converting a postfix expression into assembly language statements. In essence, for each operation encountered, this algorithm places one of the arguments in the accumulator, performs the specified operation on the second, and places the result in a temporary storage location.

As the postfix expression is scanned, the stack is used in the following manner:

- 1. If the character represents a variable, it is placed directly onto the top of the stack.
- 2. If the character represents an operator, the top two variables are removed from the stack and used as subjects for the operator. The storage location used to store the result is placed on top of the stack.

The translation of the postfix expression

$$ABC + AJK \times - / \leftarrow$$

to assembly-language code is illustrated in Table 9-1. The final assembly-language code is as follows:

```
LDA    B
ADD    C
STO    TP + 1
LDA    J
MPY    K
STO    TP + 2
LDA    A
SUB    TP + 2
STO    TP + 3
LDA    TP + 1
DIV    TP + 3
STO    TP + 4
LDA    TP + 4
STO    A
```

A couple of observations can be made:

- 1. This algorithm utilizes an excessive number of temporary storage locations. In effect, a new temporary storage location is utilized for each operator processed. Once, however, the contents of a temporary storage location have been retrieved for use by an operator, the location can be reused to store the result of another arithmetic operation.
- 2. Some extraneous instructions are generated. Specifically, the two instructions

```
STO    TP + 4
LDA    TP + 4
```

can both be omitted without changing the results.

Table 9-1 Translation from Postfix to Assembly Language Using the Algorithm in Figure 9-12

Character	<i>J</i>	Stack (after character is processed)	Code generated
A	0	A	
B	0	A,B	
C	0	A,B,C	
+	1	A, [TP + 1]	{ LDA B ADD C STO [TP + 1]
A	1	A, [TP + 1], A	
J	1	A, [TP + 1], A,J	
K	1	A, [TP + 1], A,J,K	
×	2	A, [TP + 1], A, [TP + 2]	{ LDA J MPY K STO [TP + 2]
−	3	A, [TP + 1], [TP + 3]	{ LDA A SUB [TP + 2] STO [TP + 3]
/	4	A, [TP + 4]	{ LDA [TP + 1] DIV [TP + 3] STO [TP + 4]
←	4	empty	{ LDA [TP + 4] STO A

The reuse of the temporary storage locations can be obtained by simply checking to see if either or both of the variables removed from the stack are temporary storage locations, and, if so, decrease the temporary storage index (*J*) accordingly. The flowchart for this purpose is illustrated in Figure 9-13, and the generation of assembly-language instructions is illustrated in Table 9-2. The final code is as follows:

```
LDA B
ADD C
STO TP + 1
LDA J
MPY K
STO TP + 2
LDA A
SUB TP + 2
STO TP + 2
LDA TP + 1
DIV TP + 2
STO TP + 1
LDA TP + 1
STO A
```

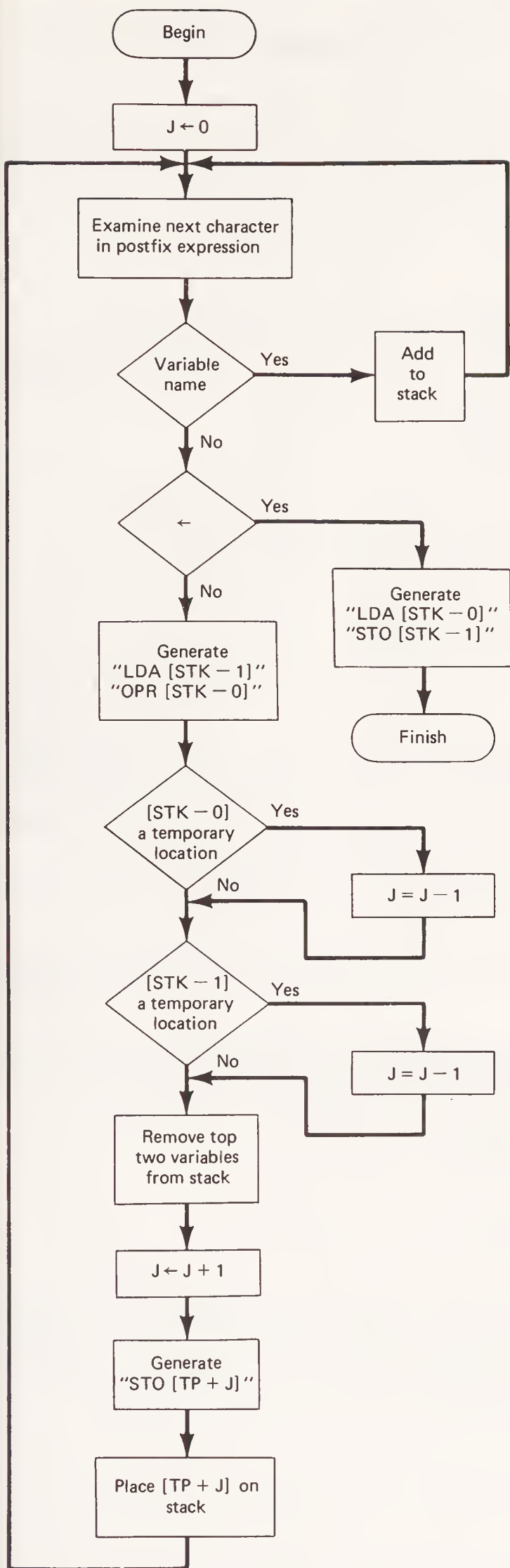


Figure 9-13 Improved algorithm for translating from postfix to assembly language

Table 9-2 Translation from Postfix to Assembly Language Using the Algorithm in Figure 9-14

Character	<i>J</i>	Stack	Code generated
A	0	A	
B	0	A,B	
C	0	A,B,C	
+	1	A, [TP + 1]	{ LDA B ADD C STO [TP + 1]
A	1	A, [TP + 1], A	
J	1	A, [TP + 1], A,J	
K	1	A, [TP + 1], A,J,K	
×	2	A, [TP + 1], A, [TP +2]	{ LDA J MPY K STO [TP + 2]
−	2	A, [TP + 1], [TP + 2]	{ LDA A SUB [TP + 2] STO [TP + 2]
/	1	A, [TP + 1]	{ LDA [TP + 1] DIV [TP + 2] STO [TP + 1]
←	0	empty	{ LDA [TP + 1] STO A

The code now uses only two temporary storage locations, whereas the previous approach used four.

As for the generation of extraneous instructions, the situation is much worse than our previous examples indicate. Suppose we translate the statement

$$A \leftarrow (B + C) \times (A + J \times K)$$

The postfix equivalent is

$$ABC + AJK \times + \times \leftarrow$$

Note that this statement is identical to the one considered previously except that the division has been replaced by multiplication and the subtraction by addition. The code generated by the algorithm in Figure 9-13 would be as follows:

```
LDA B
ADD C
STO TP + 1
LDA J
MPY K
STO TP + 2
LDA A
```

```
ADD    TP + 2
STO    TP + 2
LDA    TP + 1
MPY    TP + 2
STO    TP + 1
LDA    TP + 1
STO    A
```

An efficient programmer would accomplish the same result with the following instructions:

```
LDA    B
ADD    C
STO    TP + 1
LDA    J
MPY    K
ADD    A
MPY    TP + 1
STO    A
```

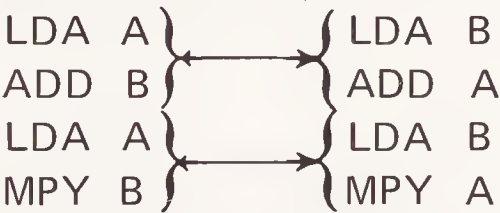
This approach requires eight instructions as compared with fourteen for the previous approach. Also, only one temporary storage location is used as compared with two for the best previous code.

To achieve this level of efficiency, two factors must be considered:

- 1. The addition and multiplication operations are commutative, that is,

$A + B = B + A$
 $A \times B = B \times A$

Thus the following operations are equivalent:



This does not hold for subtraction, division, or exponentiation.

- 2. Any time a store-accumulator instruction and a load-accumulator instruction are back to back and both refer to the same temporary location, these two instructions can be eliminated.

To illustrate the use of these points, consider the following code extracted from the previous example:

```
MPY    K
STO    TP + 2
LDA    A
ADD    TP + 2
```

316

```
STO    TP + 2
LDA     TP + 1
MPY     TP + 2
```

Using the commutative properties of the add and multiply operations, this code can be altered to

```
MPY     K
STO     TP + 2
LDA     TP + 2
ADD     A
STO     TP + 2
LDA     TP + 2
MPY     TP + 1
```

On two occasions we have back-to-back STO and LDA instructions that reference the same temporary storage location. Eliminating these gives

```
MPY     K
ADD     A
MPY     TP + 1
```

This is a significant reduction in the code.

One approach, perhaps the simplest conceptually, would be to edit the output of the algorithm in Figure 9-13. However, an algorithm to generate the efficient code directly can be developed and is given in Figure 9-14.

This algorithm differs from the one in Figure 9-13 in several respects. The major difference is that instead of automatically storing the result of an operation in the first available temporary storage location, the result is left in the accumulator and its “address,” that is, [ACC], is placed on top of the stack. The location of [ACC] within the stack must be carefully monitored. The following steps are taken:

1. If [ACC] ever gets further than one from the top, a store-accumulator instruction referencing the first available temporary storage location is generated and the address of that location replaces [ACC] in the stack. Blocks 5 and 6 in the flowchart in Figure 9-14 serve this purpose.
2. Upon encounter of an operator, the following conditions may exist:
 - a. If [ACC] is one from the top of the stack, the necessary operation instruction may be generated directly. Blocks 10 and 19 in Figure 9-14 cover this situation.
 - b. If this is not the case but (1) we have a commutative operation (+ or ×) and (2) [ACC] is on top of the stack, we can interchange the top two elements in the stack and proceed as in case a. Blocks 11, 12, and 13 take care of this.
 - c. If (a) is not true nor do we have a commutative operation but [ACC] is at the top of the stack, then we must generate a store-accumulator

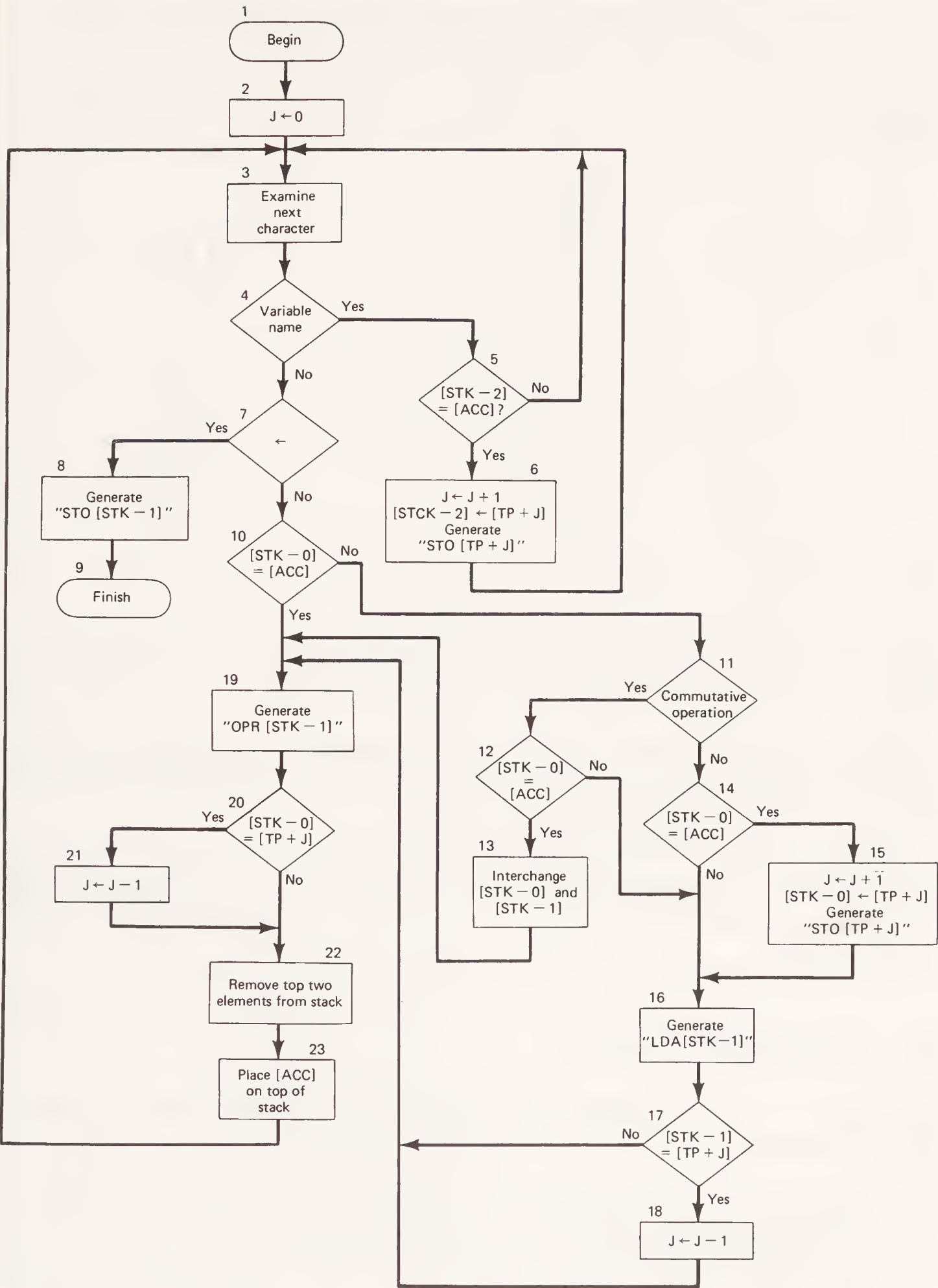


Figure 9-14 Algorithm for generating efficient code

instruction referencing the first available temporary storage location and replace $[ACC]$ by the address of this location. Blocks 11, 14, and 15 cover this situation.

- d. If $[ACC]$ is neither of the top two elements in the stack, the logic proceeds from block 10 to 11 to 14 to 16 in Figure 9-14.

Table 9-3 Translation of the Postfix Expression ABC + AJK X + X + into Assembly Code via the Algorithm in Figure 9-14

Character	<i>J</i>	Stack	Code generated
A	0	A	
B	0	A,B	
C	0	A,B,C	
+	0	A, [ACC]	{ LDA B ADD C
A	0	A, [ACC],A	
J	1	A, [TP + 1],A,J	STO [TP + 1]
K	1	A, [TP + 1], A,J,K	
X	1	A, [TP + 1], A, [ACC]	{ LDA J MPY K
+	1	A, [TP + 1], [ACC]	ADD A
X	0	A, [ACC]	MPY [TP + 1]
←	0	empty	STO A

For either case c or case d, it is necessary to first generate a load-accumulator instruction referencing the next-to-top element in the stack. Block 16 takes care of this.

Blocks 17, 18, 20, and 21 monitor the appearance of a temporary storage location in either an LDA or an operation instruction. This assures efficient utilization of the temporary storage locations.

Tables 9-3 and 9-4 illustrate the conversion of the two postfix expressions encountered in the previous discussion to assembly code. In each case, the code utilizes the minimum number of temporary locations and the minimum number of instructions.

9.6. FINISHING THE JOB

After generation of the assembly code, the remainder of the compilation process requires two steps:

- 1. Conversion of the assembler mnemonics into the proper bit patterns. This is essentially a table look-up problem.
- 2. Inserting actual addresses in place of variable names and temporary storage references.

In a practical computer, several other considerations are important. For example, the special requirements of the multiplication, division, and exponentiation operations usually must be included, as pointed out at the beginning of the last section. Furthermore, we have said nothing about the unary minus, which must be treated in a different manner (see exercises). Finally, modern computers generally have multiple registers that can be used as accumulators. By

Table 9-4 Translation of the Postfix Expression ABC + AJK X - / = into Assembly Code via the Algorithm in Figure 9-14

Character	J	Stack	Code generated
A	0	A	
B	0	A,B	
C	0	A,B,C	
+	0	A, [ACC]	{ LDA B ADD C
A	0	A, [ACC], A	
J	1	A, [TP + 1], A,J	STO [TP + 1]
K	1	A, [TP + 1], A,J,K	
X	1	A, [TP + 1], A, [ACC]	{ LDA J MPY K
-	1	A, [TP + 1], [ACC]	{ STO [TP + 2] LDA A SUB [TP + 2]
/	0	A, [ACC]	{ STO [TP + 2] LDA [TP + 1] DIV [TP + 2]
←	0	empty	STO A

efficiently utilizing these registers, most of the store and load instructions can be eliminated for the typical expression.

EXERCISES

9-1. Translate the following expressions from infix to postfix by first drawing a tree and then rearranging the branches.

- A X B/C + D/F ↑ G
- (R + Q/A) ↑ (B + G)/X - A
- X/A + G X (B - C ↑ A)
- A X (B - A X (G - X) + D)
- (G X A - (B - (D X C + G) X X) + R)/Z

9-2. Translate the expressions in the previous exercise from infix to postfix without drawing a tree.

9-3. Suppose the current value of six variables is as follows:

- A = 2 D = 3
- B = 5 E = 2
- C = 1 F = 1

320 Evaluate the following postfix expressions:

```
ABC - F/X
AB/E + F/A X B +
FABE + DC X - //
DAECDD X + / - /
```

9-4. The flowchart in Figure 9-7 assumes that the infix expression contains no errors, which is not always a good assumption. In this exercise, we shall only be concerned with the following erroneous conditions:

- 1. Expression begins or ends with an operator (recall that the unary minus was not permitted).
- 2. Two operators occur back to back.
- 3. Two variables appear back to back (recall that variable name is restricted to one alphabetic character).
- 4. An open parenthesis is followed by an operator.
- 5. A close parenthesis is preceded by an operator.
- 6. The parentheses are not matched.

In performing this error checking, the expression can be preprocessed by a routine prior to the translation routine in Figure 9-7, or the error checking can be incorporated into the flowchart in Figure 9-7. Of these, the preprocessing is probably preferable. Prepare the flowchart for this routine.

9-5. For the flowchart in Figure 9-7 and for other flowcharts in the chapter, it was assumed that the expression did not contain a unary minus sign. For example, the following statement contains two unary minus signs:

$$A \leftarrow - (X + Y) / (- C \times (A + B))$$

One method for processing the unary minus sign is to treat it as a special symbol different from the usual minus sign and assign it a level in the hierarchy above addition and subtraction but below multiplication and division. This special symbol will designate negation, and for lack of no better symbol, we might use the ϕ . In fact, this symbol could be readily inserted into the expression in place of the unary minus by the preprocessing routine described in the above exercise.

Using this approach, translate the above expression from infix to postfix. What changes are needed in the flowchart in Figure 9-7?

9-6. Suppose the assembly language instruction to negate is designated by the mnemonic NEG. This instruction simply negates the current contents of the accumulator, and makes no references to memory. Modify the flowchart in Figure 9-14 so that this operation can be included in the postfix expression being processed.

9-7. Another extension of the flowchart in Figure 9-7 is to permit functions to be used in the expression. In this problem, we shall restrict the functions to having only one argument and shall require that the name begin with the letter F

followed by a single letter of the alphabet. For example, the following statement illustrates the use of function FC:

$$A \times B + FC(X + A/G) \times C$$

In order to prevent ambiguity in the postfix expression, we might assume that this expression is preprocessed as in Exercise 9-4 and the F replaced by a special symbol such as the \$.

In many respects, the function can be treated as a special operation in translating the expression from infix to postfix. For example, the infix notation $FC(X)$ might be expressed in postfix as $X\$C$. Devise a procedure for translating general expressions containing functions from infix to postfix, and modify the flowchart in Figure 9-7 accordingly.

9-8. In translating a postfix expression containing a function into assembly language, suppose the function FC (or \$C) is called as follows:

CAL FC

This instruction transfers control to routine FC, which assumes that the accumulator contains the value of the argument at the time the CAL instruction is executed. Upon completion of the function evaluation, the accumulator contains the functional value, and control is returned to the statement immediately following the CAL.

Modify the flowchart in Figure 9-14 to encompass this situation.

10

BUSINESS SYSTEMS

In terms of numbers of applications, the use of computers in the area of business and government undoubtedly surpasses that of all other areas combined. In fact, it is in this area that the average person most frequently encounters the computer. For example, virtually every bank maintains checking accounts with the aid of a computer. The Internal Revenue Service audits tax returns with the aid of a computer. Airline reservation systems are computer-based. Universities use computers in a variety of ways, including the maintenance of the academic and financial records of the students. Many large universities use the computer in the class registration process, an operation that we shall use as our example later in this chapter.

Perhaps the distinguishing feature of business problems is that they generally involve the processing of large quantities of data. For example, a large university must maintain academic records on each of its students, numbering upwards of 40,000 or more. In addition, student financial data must be maintained separately or integrated with the academic records. Data must be maintained on each employee, on each department, on the university's budget, and so on. The organization and maintenance of these data is in itself a significant undertaking. In the first section, we shall examine the organization of these data, followed by an introduction to the operations necessary to maintain these data current.

Since the solutions of most business problems involve processing of such large quantities of data, the input/output aspects of the problem generally take precedence over the computational aspects. For example, the solution of a problem may depend almost entirely upon the time required to read a magnetic tape and be virtually independent of the amount of actual computations involved in the operation.

To achieve reasonable efficiency in business data processing, the total system must usually be considered in arriving at the best approach for accomplishing a given task. This gives rise to the idea of the *system chart* to represent the various operations that must be performed. In many cases, several programs and several

runs may be required in order to complete the total project. This aspect of business data processing is discussed in this chapter.

In earlier chapters, we have touched upon several topics relevant to business data processing. Sort algorithms such as described in Chapter 4 are used extensively in business computing. Since those sort algorithms described in Chapter 4 operate on data stored in memory, they are called *internal sorts*. Many business problems, however, involve such large quantities of data that internal sorts are impractical, thus necessitating the use of auxiliary storage units and the *external sort* algorithms such as described in this chapter. The binary-search algorithm discussed in Section 4.4 also finds considerable use in business data processing.

10.1. ORGANIZATION OF DATA

Whenever large quantities of information are involved, care must be exercised in arranging and maintaining the data in such a manner that they will be readily accessible to programs that must use them.

The accumulation of all the information required in the solution of a business-type problem (for example, student registration) is called the *data base*. The data base in turn consists of a collection of *files* containing a related subset of the data. For example, the data base for student registration may contain such files as student academic records file, class section file, classroom file, academic personnel (instructor) file, and class requests file. Technically, the use of the term *file* in this context actually designates a *logical file* as opposed to a *physical file*, the latter term designating such devices as card readers and magnetic tapes.

The data in each file are further subdivided into *records*. For example, a record in the student academic records file would contain the current academic data on a student who is currently enrolled or who has been admitted for the coming term. In some cases, the data in a file may be subdivided into records in different ways. For example, the data in the class requests file may be arranged into records containing the class requests of a given student *or* into records containing the students who have requested a given section or class. Which of these is preferable depends upon how these data are to be used. The term *record* as used here is synonymous with the term *node* used in Section 4.9.

The data in each record are further divided into *items*, or *fields*. For example, a student's academic record may consist of the following items:

1. Student's name
2. Student's social security number
3. Student's local address
4. Student's home address
5. Year (freshman, sophomore, etc.)
6. College

- 7. Major
- 8. Credit hours taken
- 9. Credit hours passed
- 10. Quality points earned
- 11. Etc.

The individual items are often subdivided into *subitems*. For example, the student's home address may be divided into the following subitems:

- 1. Street address
- 2. City
- 3. State or country
- 4. Zip code

This division may continue to obtain sub-subitems.

The general organization of the information in the data base is given by the tree in Figure 10-1.

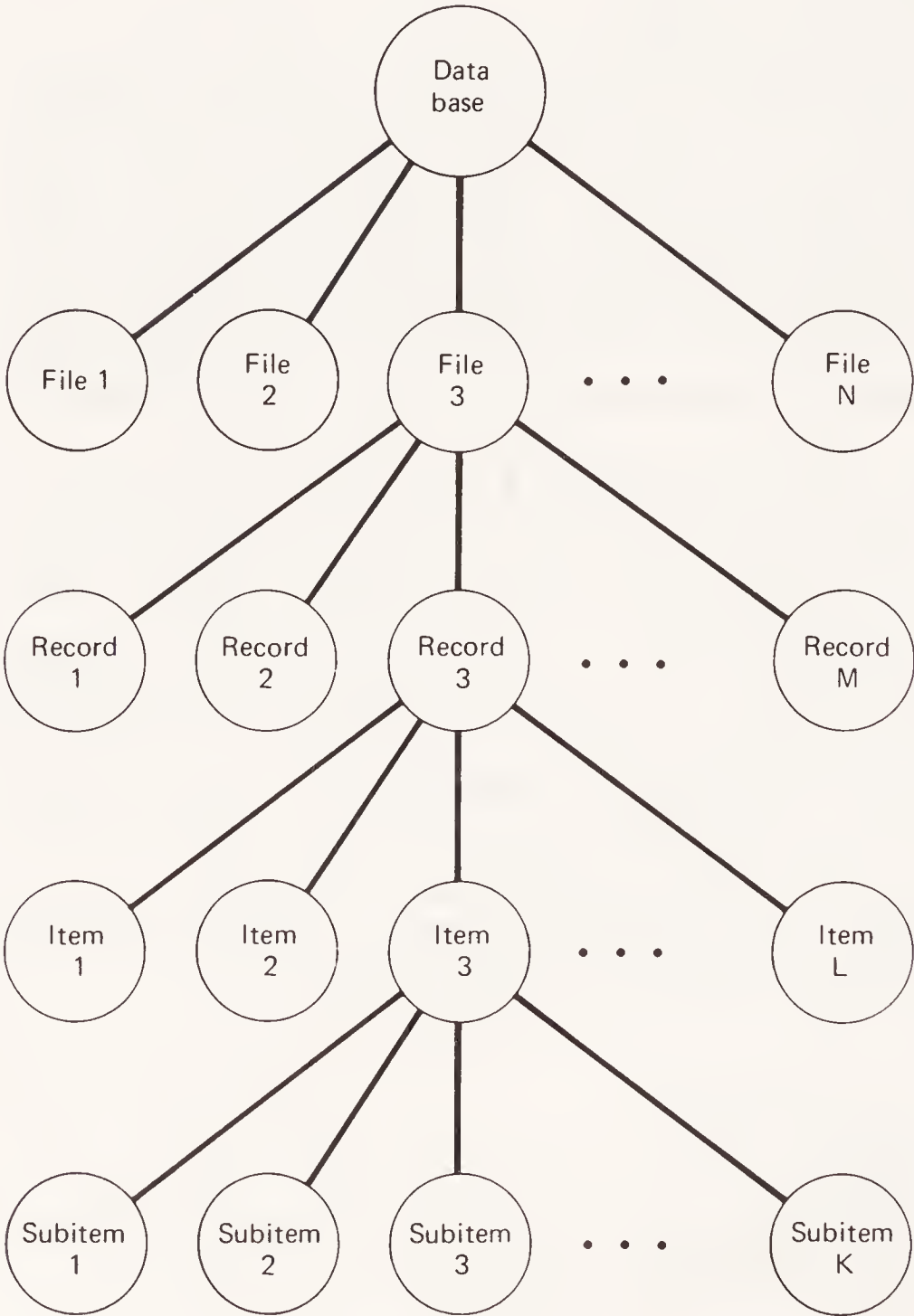


Figure 10-1 Organization of data

The records in a file may be arranged either in *sequential order* or in *nonsequential order*. When arranged in sequential order, the physical sequence in which they are arranged in the file is significant. For example, the records in the student academic records file may be arranged in alphabetical order of the students' names. The arrangement of a sequential file for eight students is shown in Figure 10-2a.

In a nonsequential arrangement, the physical sequence of records in a file is not significant. Instead, the logical sequence is maintained by an item within the record that serves as a *pointer* to the next record in the logical sequence. Figure 10-2b gives an example of a file arranged in nonsequential order. Within each record, a pointer is included. In addition, a pointer to the first record in the file must also be maintained. A record with a zero pointer indicates the last logical record in the file.

There are advantages and disadvantages to sequential files versus non-sequential files. For example, since the pointer must be included in each record of a nonsequential file, it will require more storage area than a sequential file. Suppose, however, a record for GENE DAVIS is to be added to the file. This record should appear immediately after SUE DANIELS in the logical sequence. In a sequential file, insertion of this record would require that the records beginning with JANE FOSTER and continuing through the remainder of the file must be shifted down. In the nonsequential file, however, the record for GENE DAVIS may be added to the end of the file. The pointer of the immediately preceding logical record is transferred to the record being added, and the pointer of the immediately preceding logical record is set to point to the record being added. Figure 10-3 illustrates the changes made in adding the record GENE DAVIS to the original nonsequential file. In general, the addition (and deletion) of records from nonsequential files is substantially easier than with sequential files.

1	JOHN ADAM
2	JOE BLEAUX
3	SUE DANIELS
4	JANE FOSTER
5	DAVID JONES
6	NANCY MILLER
7	DAN PORTER
8	JOHN SMITH

a. File arranged in sequential order

	Name	Pointer
1	DAVID JONES	8
2	JOE BLEAUX	7
3	JANE FOSTER	1
4	JOHN SMITH	0
5	JOHN ADAM	2
6	DAN PORTER	4
7	SUE DANIELS	3
8	NANCY MILLER	6

First Record Pointer: 5

b. File arranged in nonsequential order

Figure 10-2 Arrangements of records in files

1	DAVID JONES	8	1	DAVID JONES	8
2	JOE BLEAUX	7	2	JOE BLEAUX	7
3	JANE FOSTER	1	3	JANE FOSTER	1
4	JOHN SMITH	0	4	JOHN SMITH	0
5	JOHN ADAM	2	5	JOHN ADAM	2
6	DAN PORTER	4	6	DAN PORTER	4
7	SUE DANIELS	3	7	SUE DANIELS	9
8	NANCY MILLER	6	8	NANCY MILLER	6
			9	GENE DAVIS	3

Before

After

Figure 10-3 Addition of a record to a nonsequential file

The choice between sequential and nonsequential files also depends upon the nature of the auxiliary device employed. For example, magnetic tape units and card readers are inherently sequential units; that is, information is read in sequence. Therefore, nonsequential files cannot be practically implemented on these units. With disks or drums, however, files can be implemented either sequentially or nonsequentially. The disk unit is capable of retrieving the record at a certain location, much like the contents of a word at a specific address can be retrieved from memory. Therefore, records can be read from a disk or drum in any order, regardless of their physical or logical order. This is termed *random access*, and files so implemented are called *direct-access files*.

In many systems, data can be entered directly. For example, in a student registration system, it would be possible for the student or operator to enter class requests directly into the system. Some checking may also be accomplished, such as the existence of the specific section so requested. The term *on-line* applies to systems that permit the entry of data in this fashion.

In the above example, the checking of the entry was to insure only that the request was valid. The system did not ascertain if any space was actually available in the requested section. Many registration systems accumulate all requests and then allocate seats in classes at a later time, using some priority system as a basis for deciding who gets first shot at a seat.

At least a few registration systems, however, go the additional step of actually allocating seats at the time the requests are made. For example, suppose the student requests a seat in Basket Weaving 119, Section 4, at 10:30 MWF. This is a popular course offered at a popular time and fills up early. If a seat is available, the system allocates it to the student. If a seat is not available in this particular section, it might display what sections of this course have available seats. If one of these sections is satisfactory, the student changes his request to obtain an

alternate section. If none are satisfactory, the student must choose another course.

In the system described in the above paragraph, the student enters information, the computer processes this information, and returns the results to him in a matter of seconds. He can then alter his request if necessary. In this way, the computer is affecting the process (that is, registration) that is generating the inputs. Systems operating in this manner are said to be *real-time* systems. They are also on-line systems. That is, a real-time system must be an on-line system, but the converse is not necessarily true.

In many data processing systems, the security of some data, for example, faculty salaries, must be maintained. Although this problem is more serious in on-line systems, it nevertheless exists in other systems. In effect, unauthorized individuals must be prevented from accessing certain files of data. One approach is to maintain a password or key associated with the file, and only those who know the password can gain access to the file.

Another problem that must be considered in maintaining the data base for business systems is that of backup. For example, suppose all course requests for the coming semester have been entered into a file that is implemented on magnetic tape. How many copies of this tape should be kept? To some extent, this depends upon how much effort is required to create a new copy. If the students submitted written requests which were then keypunched, a new file could be created from the punched cards. If, however, the data were entered on-line, this option is probably not open unless cards were punched as the data were entered.

What could happen to a reel of magnetic tape? It could be dropped and the edges crushed, which would probably prevent its being read. It could be scratched by the operator during the mounting process. If the tape were old, it could possibly lose part of the magnetic coating. The tape drive could malfunction and ruin the tape. It could be lost. In short, enough possibilities exist that a single copy is not sufficient. Furthermore, a fire or other catastrophe could occur at the computer center, making it desirable to maintain a copy at another location.

Unfortunately, there is no simple, direct answer to the question of how much backup should be maintained. Naturally, more backup should be maintained for the more difficult files to reconstruct.

Finally, the question of verification of data in the system should be addressed. Any results derived from inaccurate data are also inaccurate. Many systems essentially require that new data be entered by one person and checked or verified by another. This procedure has been used in keypunching for years, and a special machine called the *verifier* permits an operator to key in data which are compared to that punched on a card for consistency. Listing of changes in records may be made for subsequent checking. Deletion of records is particularly hazardous, and in some systems a file of deleted records is maintained. In short, considerable precaution must be exercised to preserve the accuracy of the data within the system.

CALL NUM	COURSE ABBREV. NUM.	SEC. NUM	CR.	TIME BEGIN-END	DAYS MTWTFSS	TEACHER	PAGE 11

CIVIL ENGINEERING							
4488	CE	156	LAB	1 3.0	1030-1130 130-0430	T TH W	WINTZ JR W
4489	CE	156	LAB	2 3.0	1230-0130 130-0430	T TH TH	HARRIS J
4490	CE	160		1 3.0	730-0830	M W F	DART JR O
4491	CE	173	LAB	1 1.0	130-0430	T	DART JR O
4492	CE	173	LAB	2 1.0	130-0430	T	MCKEE D
4493	CE	183	LAB	1 4.0	830-0930 130-0430	M W F M	POPLIN J ARMAN A
4494	CE	185	IND	1 3.0	730-0900	T TH	MODLIN D
4495	CE	189	LAB	1 3.0	1130-1230 130-0430	M W TH	DEMARCHÉ J
4496	CE	190		1 3.0	1030-1130	M W F	AGUILAR R
4497	CE	205		1 3.0	TBA		TURNER H
4498	CE	207		1 3.0	TBA		POPLIN J
4499	CE	215		1 3.0	TBA		DART JR O
4500	CE	217		1 3.0	TBA		MODLIN D
4501	CE	249		1 3.0	TBA		MCKEE D
4502	CE	285		1 3.0	TBA		AGUILAR R
4503	CE	400	IND	1 1-9	TBA		
COMPUTER SCIENCE							
4504	CSC	40		1 2.0	830-0930	M W	
4505	CSC	40		2 2.0	800-0900	T TH	
4506	CSC	40		3 2.0	300-0500	T	
4507	CSC	40		4 2.0	330-0430	M W	
4508	CSC	41		1 2.0	830-0930	M W	
4509	CSC	41		2 2.0	300-0500	T	MURRILL P
4510	CSC	49		1 3.0	930-1030	M W F	HANCHEY C
4511	CSC	49		2 3.0	230-0330	M W F	
4512	CSC	49		3 3.0	630-0800N	T TH	RUDD W
4513	CSC	51		1 3.0	930-1030	M W F	
4514	CSC	51		2 3.0	230-0330	M W F	NANCE O
4515	CSC	52		1 3.0	830-0930	M W F	RUDD W
4516	CSC	52		2 3.0	900-1030	T TH	
4517	CSC	59		1 3.0	1030-1130	M W F	
4518	CSC	62		1 3.0	1130-1230	M W F	
4519	CSC	62		2 3.0	1130-1230	M W F	
4520	CSC	62		3 3.0	1230-0130	M W F	
4521	CSC	62		4 3.0	130-0300	T TH	
4522	CSC	70		1 3.0	1030-1130	M W F	HANCHEY C
4523	CSC	70		2 3.0	630-0800N	M W	SMITH C
4524	CSC	80		1 3.0	900-1030	T TH	SMITH C
4525	CSC	101		1 3.0	1030-1200	T TH	RUDD W
4526	CSC	102		1 3.0	930-1030	M W F	TYLER J
4527	CSC	154		1 3.0	1230-0130	M W F	BREWEN J
4528	CSC	199		1 1-3			
CONSTRUCTION TECHNOLOGY							
4529	CTCH	1		1 1.0	130-0230	M	HAND J E
4530	CTCH	6	LAB	1 2.0	830-1030	T TH	BAKER O J
4531	CTCH	6	LAB	2 2.0	1030-1230	T TH	HAND J E
4532	CTCH	6	LAB	3 2.0	330-0530	T TH	OIVANKI
4533	CTCH	108		1 3.0	TBA		
4534	CTCH	113		1 2.0	400-0600	W	TOUPS R
4535	CTCH	119	LAB	1 2.0	930-1030 930-1130	T TH	OIVANKI
4536	CTCH	119	LAB	1 2.0	930-1030 130-0330	T TH	OIVANKI
4537	CTCH	151	LAB	1 4.0	1030-1130 1230-0330	T TH M W	HERES H

Figure 10-4 Page from schedule of classes booklet (Courtesy Louisiana State University)

10.2. PROCESSING OF FILES

In this section, we shall use as our example typical runs that may comprise portions of a student registration system. Basically, we shall assume that we begin with a Student Request File that contains the following information:

- 1. Student ID number
- 2. Line numbers of courses (maximum of 10) that the student wishes to take and the credit hours of each

In addition to this file, we have a Class Section File that contains the following:

- 1. Line number of the course
- 2. Department offering course
- 3. Course number
- 4. Section
- 5. Credit hours
- 6. Time offered
- 7. Instructor
- 8. Room

In effect, the information in the Class Section File is made available to the students in the form of a class schedule booklet. (See Figure 10-4.) From this, the student submits his course requests (Figure 10-5), from which the student’s name, the courses he selects, and the credit hours of each course are punched. The number of credit hours is punched because some courses may be carried for variable credit.

STUDENT NAME		STUDENT NUMBER	CLASSIFICATION	BIRTH DATE	S.V.R.	DATE
LOCAL ADDRESS OR DORMITORY		LSU BOX	STUDENT COURSE REQUEST			
			(PLEASE PRINT)			
CITY, STATE		ZIP	TELEPHONE	CONSULT INSTRUCTIONS BEFORE ATTEMPTING TO PREPARE THIS FORM		
HOME STREET ADDRESS				CALL NUMBER	DEPARTMENT (ABBREV.)	COURSE NUMBER
CITY, STATE		ZIP		SECTION NUMBER	CREDIT HOURS	TIME START- END
NAME OF PARENT OR GUARDIAN OR EMERGENCY						DAYS (CIRCLE)
PARENT'S STREET ADDRESS						M T W T F S
CITY, STATE		ZIP				M T W T F S
STUDENT SEX-MARITAL STATUS		1 2 3 19				M T W T F S
PLACE OF EMPLOYMENT - HOURS PER WEEK		DEC MAY AUG				M T W T F S
LOUISIANA STATE UNIVERSITY		SCHEDULE APPROVED:		TOTAL HOURS		REGISTRAR'S COPY
BATON ROUGE CAMPUS		ADVISOR		DEAN		

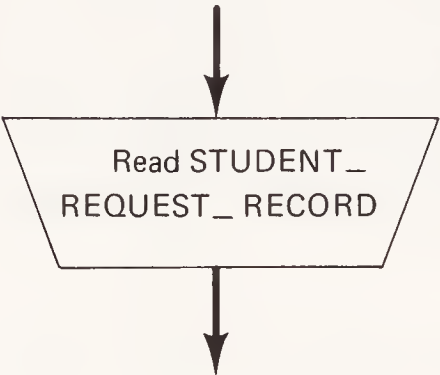
Figure 10-5 Course request form (Courtesy Louisiana State University)

330 In processing files with such languages as PL/I and Cobol, the basic input/output operations apply to records, not to the individual items or subitems. That is, suppose each record in the Student Request File is defined or *declared* as follows:

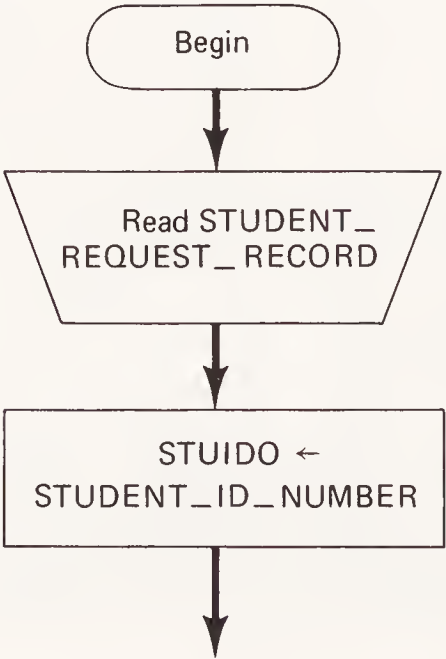
```
STUDENT_REQUEST_RECORD
01  STUDENT_ID_NUMBER
01  SECTION_REQUEST OCCURS 10 TIMES
    02  LINE_NUMBER
    02  CREDIT_HOURS
```

In this declaration, we have adopted the PL/I convention of permitting a variable name to include the underline character, `_`. The record name is `STUDENT_REQUEST_RECORD`. Each record contains eleven items, the first of which is the `STUDENT_ID_NUMBER` and the last ten of which are the `SECTION_REQUEST`'s (note the `OCCURS 10 TIMES` clause in the above declaration). Each of the ten `SECTION_REQUEST` items contains the subitems `LINE_NUMBER` and `CREDIT_HOURS`.

To read a record from the Student Request File, we shall use the following trapezoidal-shaped input block:



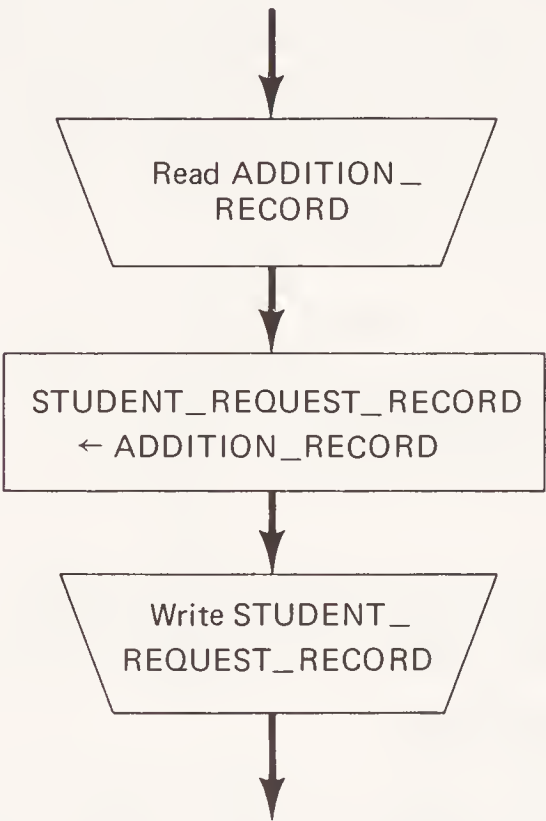
This instructs the computer to read an entire record from the Student Request File. All items and subitems appearing within the respective record are automatically assigned new values as a result of this read operation. For example, the following operations are entirely legitimate:



A record is read from the Student Request File, and the variable STUIDO is assigned the value of the student ID number appearing within this record.

Write operations are treated in the same manner as read operations, and the trapezoidal-shaped block will be used to designate both input and output operations in our flowcharts in this chapter.

We will also permit our assignment operator to apply to records as well as items. For example, suppose we have a file called the Additions File for which the record is named ADDITION_RECORD. The arrangement of items and subitems within ADDITION_RECORD is identical to that within STUDENT_REQUEST_RECORD. Now suppose that we would like to copy a record from the Addition File onto the Student Request File. The appropriate flowchart is as follows:



In this flowchart, the assignment operator has been used to copy ADDITION_RECORD into STUDENT_REQUEST_RECORD. Alternatively, assignment statements can be used for the individual items or subitems within the records.

As our first example of a typical data-processing run, suppose we have created a Student Request File and now want to add some late registrants to this file. Their requests have been punched on cards consistent with the data format of the Student Request File. For reasons which should become more evident later in this section, such files generally need to be sorted in some manner. Therefore, let's assume that the records in our Student Request File are in ascending order of the student ID numbers. We begin with a sorted file, and after the additions are incorporated, we want the resulting file to also be arranged in ascending order of the student ID numbers.

One approach is to simply add the late registrants to the end of the Student Request File and then sort the entire file. This is not, however, the most efficient approach, since the late registrants can be added in such a manner that the resulting file is arranged as desired.

In order to use the following approach, the additions must be sorted to arrange them in ascending order of the student ID numbers. Since the additions are generally substantially fewer in number than the records in the file, sorting the additions is much easier than sorting the entire file. We might call the sorted additions the Addition File.

In effect, the desired run will combine, or *merge*, the Addition File with the Current Student Request File and produce an Updated Student Request File. The logic is as follows:

- 1. Read into memory one record from the Addition File and one record from the Current Student Request File.
- 2. If the student number in the record from the Addition File is less than the student number in the record from the Current Student Request File, write the record from the Addition File onto the Updated Student Request File and read the next record from the Addition File.
- 3. Otherwise, write the record from the Current Student Request File onto the Updated Student Request File and read the next record from the Current Student Request File.

This operation involves the use of three files:

- 1. Current Student Request File
- 2. Updated Student Request File
- 3. Addition File

We shall declare the records within these files as follows:

Current Student Request File:

```
CURRENT_SR
01  STU_ID
01  COURSE_REQUEST OCCURS 10 TIMES
    02  LINE_NUMBER
    02  CREDIT_HOURS
```

Updated Student Request File:

```
UPDATED_SR
01  STU_ID
01  COURSE_REQUEST OCCURS 10 TIMES
    02  LINE_NUMBER
    02  CREDIT_HOURS
```

Addition File:

```
A_RCD
01  STU_ID
01  COURSE_REQUEST OCCURS 10 TIMES
    02 LINE_NUMBER
    02 CREDIT_HOURS
```

Although items within the respective records are given the same name, these names do not apply to the same area in memory. In effect, sufficient area within memory is reserved to store each of the records. Therefore, the name STU_ID appearing in, say, an assignment statement would be meaningless, since it could apply to the STU_ID item within any one of the three records. To remove this ambiguity, a modifier is attached. For example, STU_ID OF A_RCD designates the STU_ID item appearing in the area reserved for the record A_RCD.

The flowchart in Figure 10-6 incorporates the logic required to merge two files. Blocks 1 and 2 read one record from the Addition File and one record from the Current Student Request File. Their respective student ID numbers are compared in block 3. If the student ID number in the record from the Addition

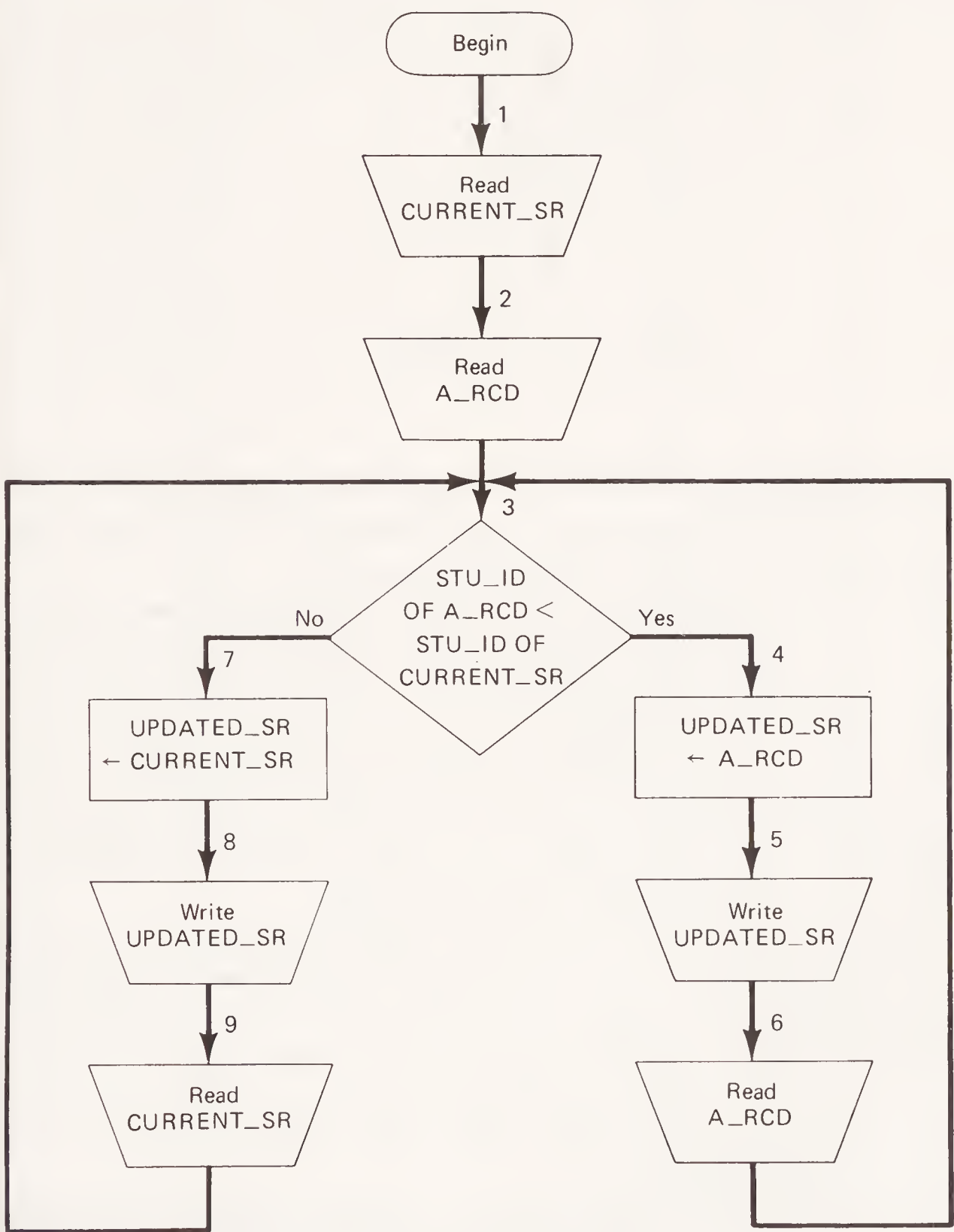
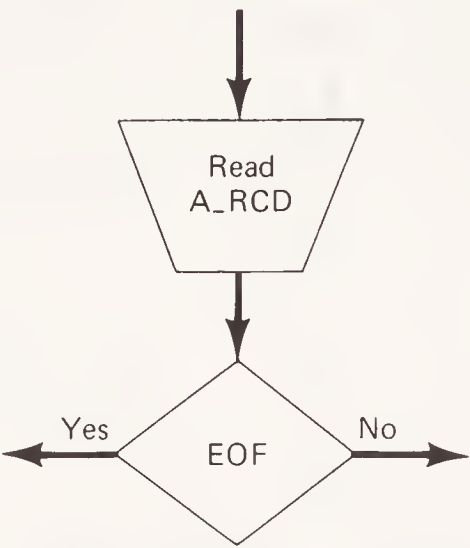


Figure 10-6 Flowchart for merging two files

File is the lower of the two, A_RCD is copied into UPDATED_SR in block 4, UPDATED_SR is written in block 5, and a new record is read from the Addition file in block 6. If the student number in the record from the Current Student Request File is the lower of the two, blocks 7, 8, and 9 perform the corresponding operations.

The flowchart in Figure 10-6 functions very nicely until the end of one of the files is encountered. Whenever any file is created upon such a physical medium as a magnetic tape, the file is terminated with a special message that indicates end-of-file (EOF). Programming languages suitable for business data processing permit the transfer to another location within the program whenever the end-of-file is encountered. In our flowcharts, we shall use the convention illustrated by the following example:



That is, we shall permit a decision block to appear immediately after a read operation to ascertain if the end-of-file was encountered. No other blocks may appear between the read block and the EOF decision block.

In the flowchart for the merge program in Figure 10-6, when the end-of-file is encountered on one of the files, we simply want to copy the remainder of the other file onto the Updated Student Records File. The necessary flowchart is given in Figure 10-7. If the end-of-file is encountered first on the Addition File, blocks 10, 11, and 12 copy the remainder of the Current Student Request File onto the Updated Student Request File. If the end-of-file is encountered first on the Current Student Request File, blocks 13, 14, and 15 copy the remainder of the Addition File onto the Updated Student Request File.

As was mentioned briefly in the last section, maintaining the accuracy of the information in the data base is a formidable task. One problem is that in even the most carefully designed systems, the records in files sometimes get out of sequence. In some cases, a special program may be written just to check the records in a file to be sure they are in the proper order. In other cases, such checking can be incorporated into other runs, thus avoiding an extra run and another reading of the same file.

As an example of how this can be done, we shall incorporate a sequence check into the flowchart of the merge run in Figure 10-7. To do this, one must

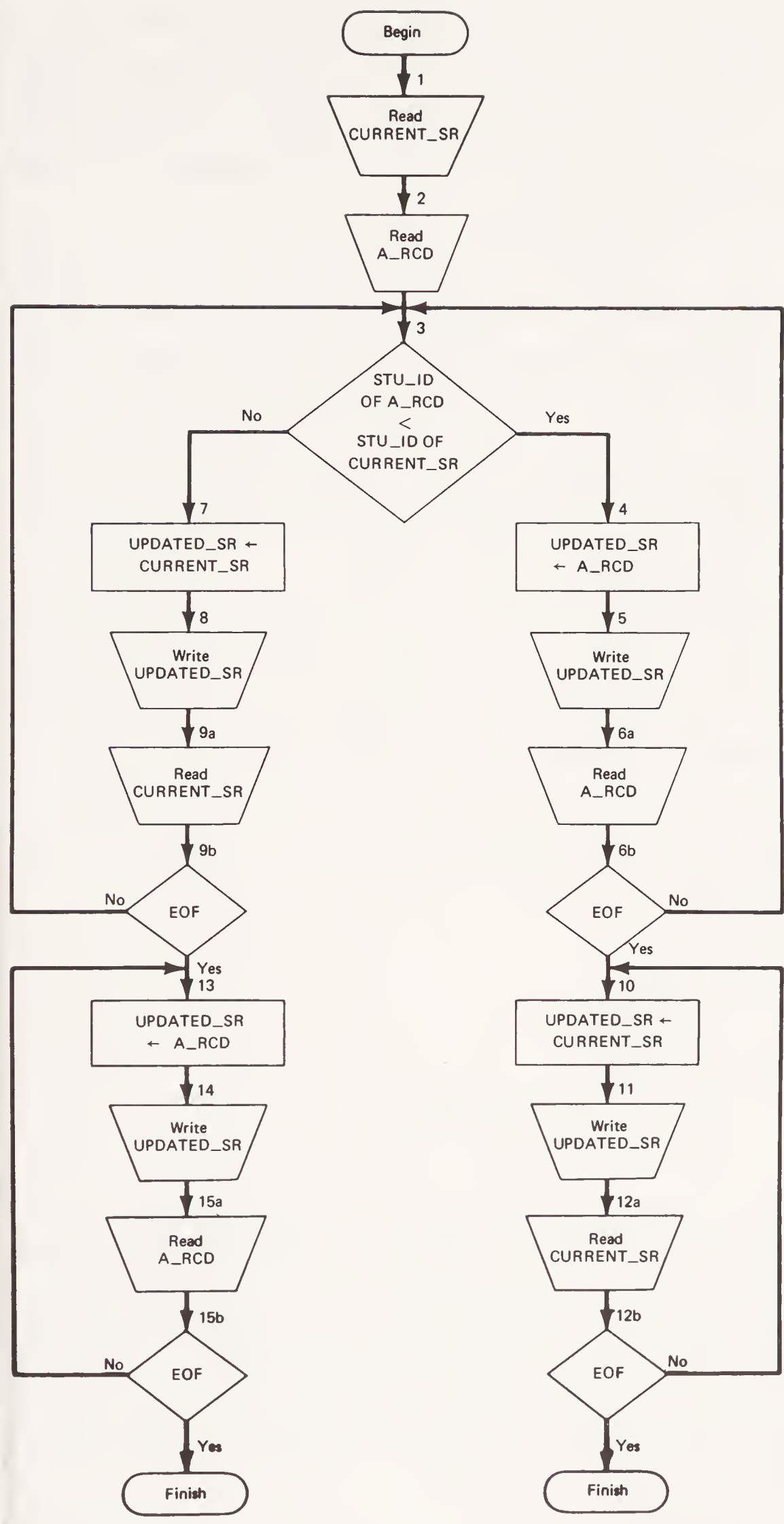


Figure 10-7 Complete flowchart for merging the Addition File with the Student Request File

first address the question of what to do when an out-of-sequence record is found. One possibility would be to print a message to the computer operator instructing him to terminate the current run, sort the current Student Request File, and restart the merge run. The Student Request File is, however, probably a large one, and hopefully only a few records are out of sequence. If possible, sorting of large files should be avoided.

An alternative is to create another file called the Out-of-Sequence Records File. The record within this file will be designated OUT_SEQ, and the declaration of the items and subitems is identical to that of the other three files. As illustrated in the flowchart of Figure 10-8, the ID number in all records (except the first) of the Current Student Request File is compared with the student ID number of the previous record. If the ID number in the current record is less than the ID number in the previous record, the current record is written onto the Out-of-Sequence Records File, another record is read from the Current Student Request File, and the comparison repeated. This is most conveniently implemented by procedure CHECK in Figure 10-8b.

What can be done with the Out-of-Sequence Records File? Since it must be merged into the Student Request File, one approach would be to first sort the Out-of-Sequence Records File and use it in place of the Addition File in the flowchart in Figure 10-8. The Updated Student Records File produced by the first run is used as the Current Student Records File in the second run.

As our final example of the processing of files, suppose we would like to create from the Current Student Request File another file called the Section Request File that contains each request of a section in a separate record. The declaration is as follows:

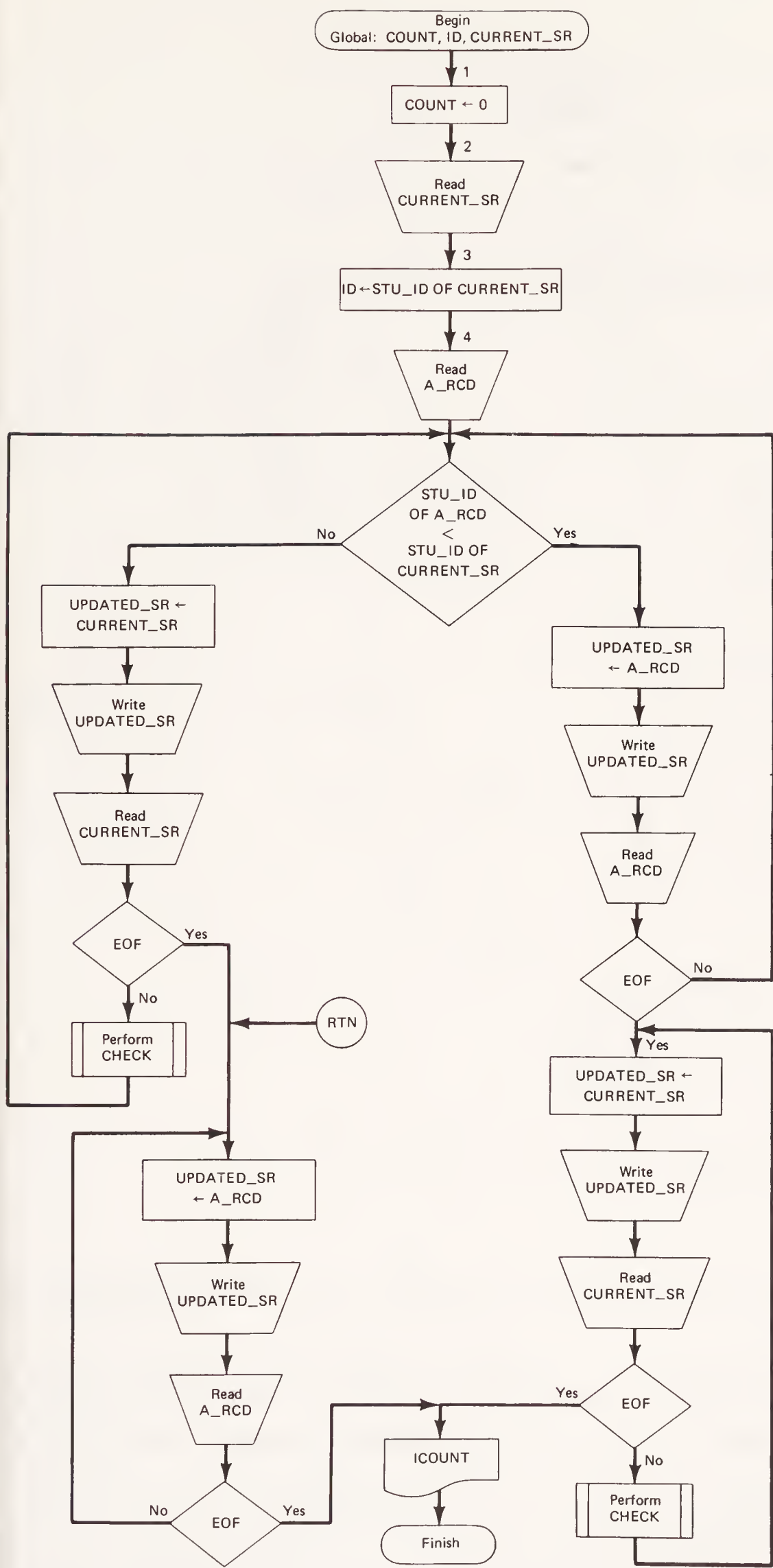
Section Request File:

```
REQ_RCD
01  STU_ID
01  REQUEST
    02  LINE_NUMBER
    02  CREDIT_HOURS
```

The declaration of the Current Student Request File is as given earlier.

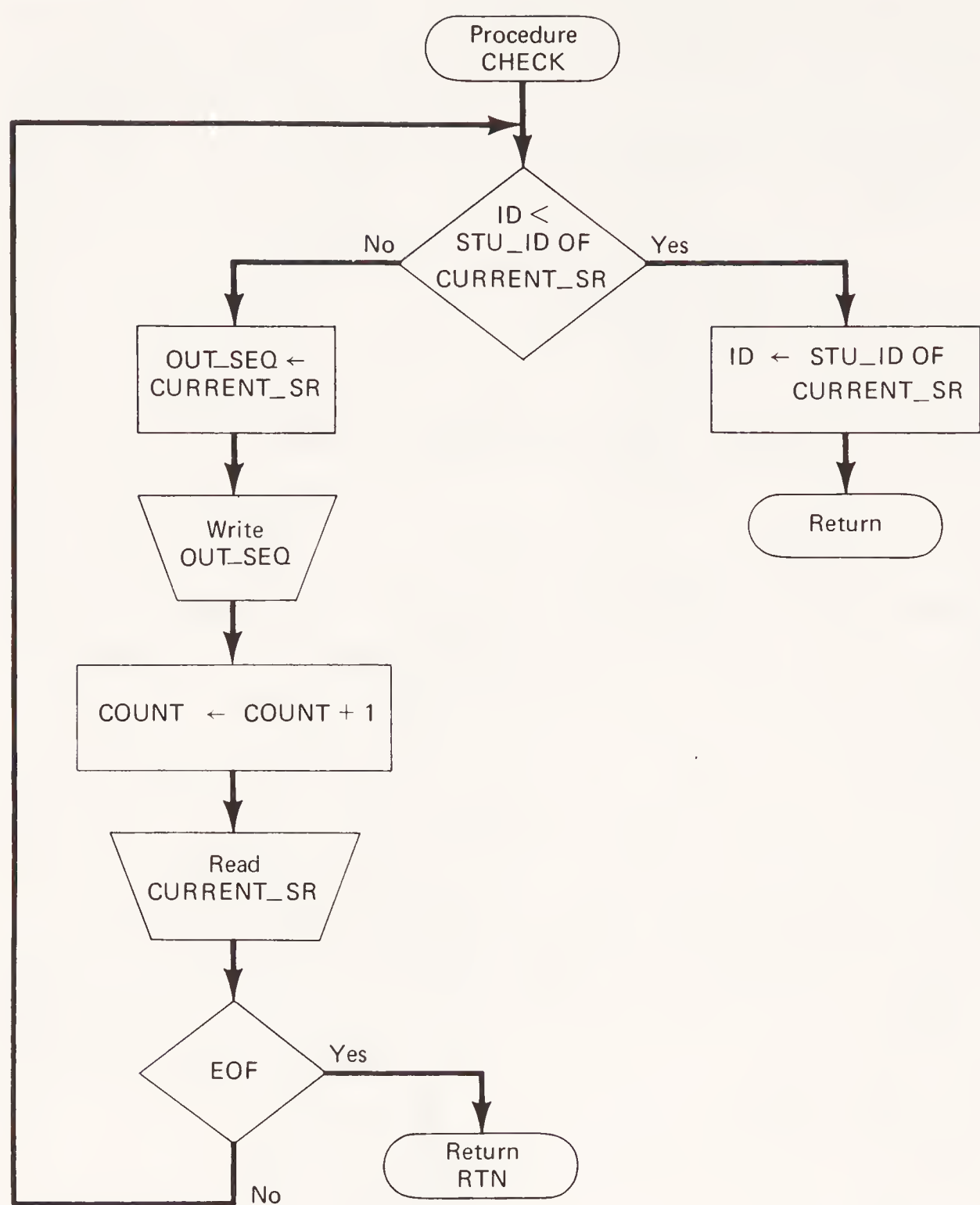
The flowchart to create the Section Request File from the Current Student Request File is given in Figure 10-9. This flowchart illustrates the creation of records by moving individual items instead of the entire record. In addition, the flowchart in Figure 10-9 indicates that the OCCURS clause in the record declaration designates the respective item as being a single-subscripted array. In block 6 of Figure 10-9, COURSE_REQUEST_j is moved to CR, which is declared as follows:

```
01  CR
    02  LINE_NUMBER
    02  CREDIT_HOURS
```



a. Main flowchart

Figure 10-8 Incorporation of sequence checking into the flowchart of Figure 10-7



b. Procedure CHECK
Figure 10-8 (Continued)

A variable is much like an item in a record in that it may be declared as containing subitems.

In the flowchart in Figure 10-9, we have assumed (refer to block 7) that an encounter of a zero line number indicates the end of this student's section requests. In doing this, we are not using the entire area reserved for each record in the file. This can be avoided by including another entry in each record designating the number of requests made by a given student, and then including only that number of requests. The records are now variable in length as opposed to fixed in length as previously defined. Unless all or almost all of the students make ten requests, the use of variable length records in this case would reduce the total length of the file.

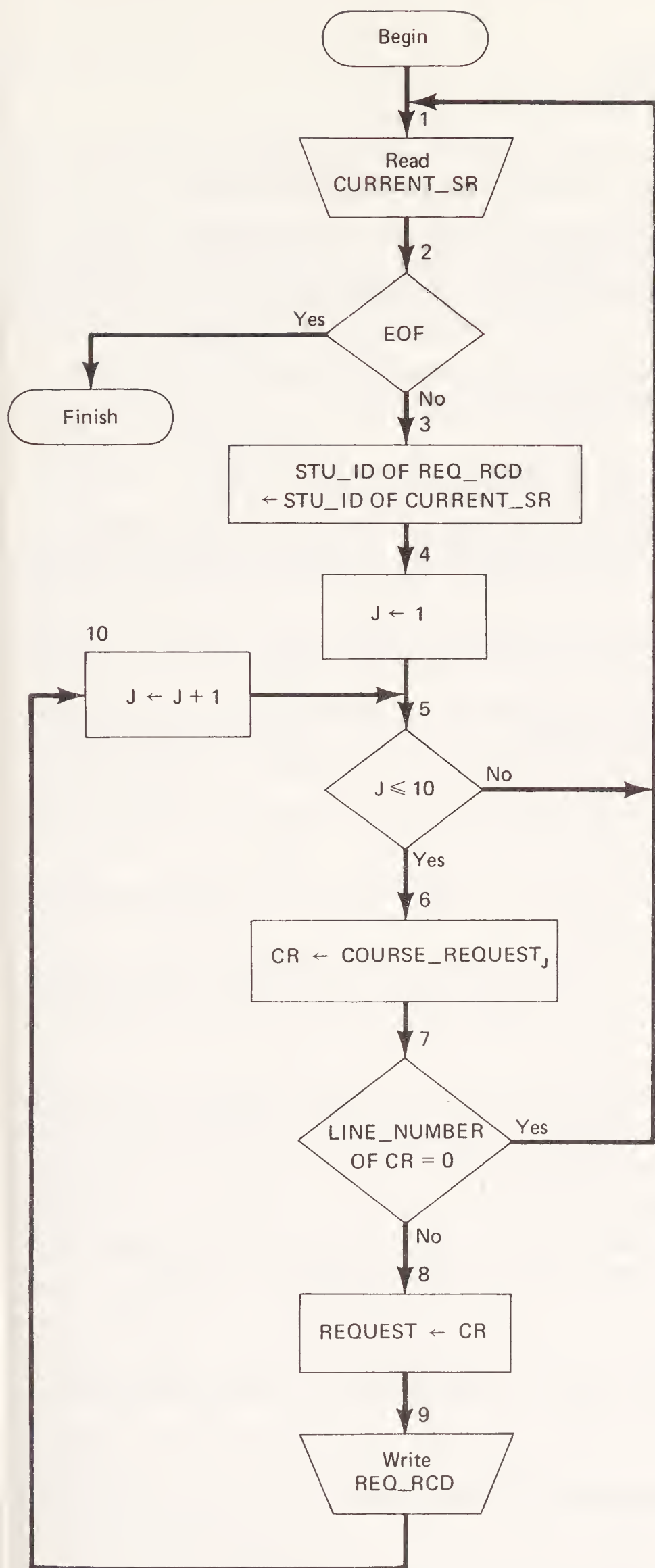


Figure 10-9 Flowchart for creation of Section Request File from Current Student Request File

10.3. SYSTEM CHARTS

Up to this point in this text, we have used flowcharts to represent the logic in an individual computer program. In business systems where more than one program is needed to accomplish a specific undertaking, the relationship between the various programs must also be represented. The system chart serves this purpose in much the same manner as the flowchart represents the logic of an individual program.

As an example, we shall develop a system chart for a portion of a registration system. The overall registration procedure is as follows:

- 1. Departments within the university submit the courses to be offered during the coming term.
- 2. The booklet giving the schedule of classes (Figure 10-4) is prepared.
- 3. Packets are prepared for all students who are enrolled during the current term.
- 4. The students complete their course request forms (Figure 10-5) and return to the registrar.
- 5. To the extent possible, students not enrolled this term but planning to enter the university the coming term are given the opportunity to register.
- 6. A report is sent to the departments giving the number of students who requested each section.
- 7. The departments may, at their discretion, add or delete sections in response to student demand.
- 8. The scheduling run is made where the seats are allocated to the students.
- 9. A walk-through registration is conducted for those students who failed to preregister.

In reality, this sequence of steps is far simpler than a true registration system. We shall be concerned only with the first seven steps and thus only indicated the last two steps in a general manner. The program, called the scheduler, to accomplish step 8 is quite an undertaking in itself.

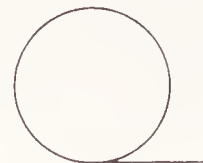
Figure 10-10 gives the system chart for steps 1 through 7 of the registration procedure outlined above. In general, the use of the boxes is as follows:



Rectangular boxes indicate processing operations.



Card-shaped boxes indicate files of data on punched cards.



Circular boxes of this type indicate files of data on magnetic tape.



This box indicates files of data as printed documents or forms.

As will be noted from several of the rectangular boxes in Figure 10-10, noncomputer operations are also included in the system chart. For example, boxes 1, 7, 14, and 17 designate keypunch operations, box 6 denotes the act of registration on the part of the students, and box 13 denotes action by the academic departments. These boxes are as much an integral part of the total process as the computer runs themselves.

The two programs whose flowcharts were developed in the last section are included in Figure 10-10. The flowchart in Figure 10-9 is the basis for the program to perform block 10, the preparation of the Individual Section Request File from the Student Request File. If the computer's memory were large enough, a storage location could be reserved for each section of each course and the tally of the requests for each could be made directly from the Student Request File. The system chart in Figure 10-10 assumes that this is not possible. To accomplish the tally, the file of individual section requests is prepared in block 10 and then sorted on line number in block 11. The sorting operation will group together all requests for the same section, making the tally of the requests by block 12 quite easy. Since the groups appear in order of the section line number, the result of the tally can be printed with the additional data from the Section Offering File to make a very readable document to be sent to the departments (see Figure 10-11).

The flowchart in Figure 10-8 forms the basis for the program for block 20. Note the line connecting the Out-of-Sequence File with the Additional Request file with an intermediate sort. This run should be repeated until no out-of-sequence records are obtained. Two runs should be sufficient.

The program for block 16 also deserves attention. This program incorporates the changes in section offerings submitted by the departments into the Section Offering File. Suppose a department proposes to change or delete a nonexistent course? What if they propose to add a course at an unreasonable hour? In effect, there are several erroneous operations that could be attempted. It is certainly not practical to terminate the run each time such a situation is encountered. Instead, each erroneous change is printed (or placed in an Error File and subsequently printed) and the run continued. Upon completion, all erroneous changes are forwarded to the Registrar's office, where they are corrected and resubmitted to be processed in another run. The success of the total system often depends upon how effectively the system copes with the meaningless and ridiculous data that enters.

Of course, changes can be entered that are valid in the sense that they can be made but on the other hand were not meant to be made by the department. In many cases, a verification step can be invaluable. For example, a preliminary copy of the portion of the class schedule booklet pertaining to each department

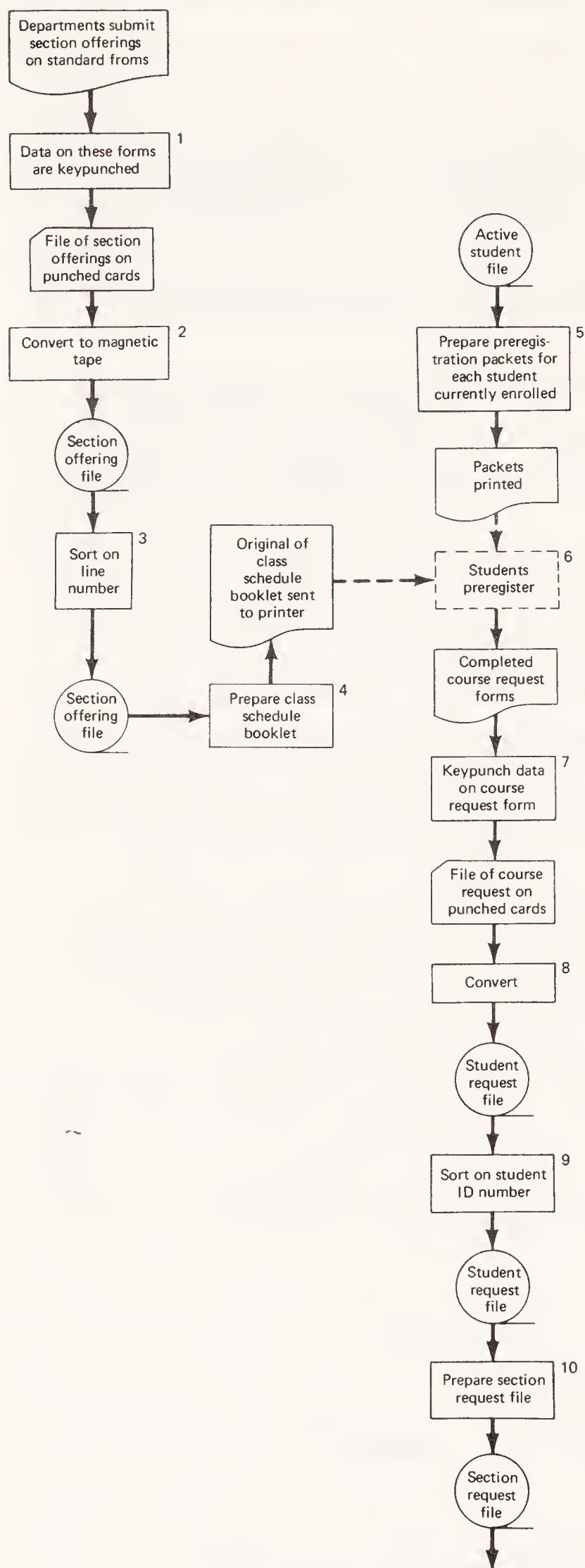


Figure 10-10 System chart for preregistration procedure

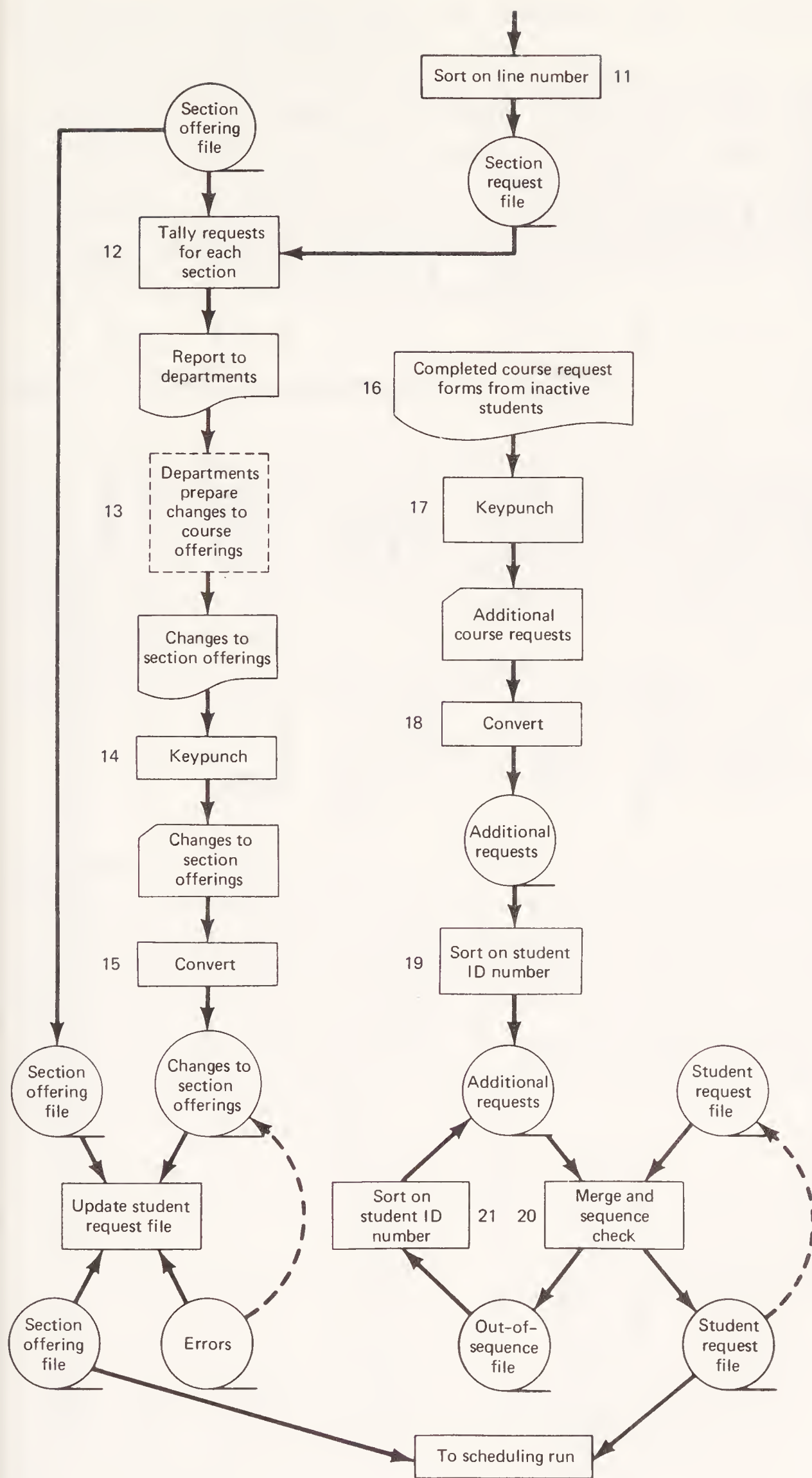


Figure 10-10 (Continued)

* * * * * M A N A G E M E N T R E P O R T * * * * *										PAGE
(PLEASE PRINT ALL CHANGES IN RED - DO NOT TYPE)										
07/02/72										
COMPUTER SCIENCE										
CSC -07										

could be sent to the department for verification prior to printing the final document to be sent to the printer.

The preparation of the system chart for a particular system is referred to as *system design*. This endeavor is challenging enough when a system is considered by itself but becomes even more challenging when this system must be integrated with other systems for the particular organization. For example, the payment of fees during registration means that the registration system interacts with the financial system and for most successful results the interaction between these two systems must be carefully considered.

10.4. EXTERNAL-SORT ALGORITHMS

In the system chart in Figure 10-10, the sorting of large files of data was called for in several boxes. For files such as the Student Request File, there is insufficient area in the computer's internal memory to store even a small fraction of the file that would be encountered at a university with 20,000 students. Therefore, we are forced to resort to using sort algorithms that operate on external files.

To illustrate the logic involved in an external sort, suppose we begin with the following list of numbers to be arranged in ascending order:

1214, 1796, 1417, 1906, 1047, 1814, 1215,
1516, 1604, 1108, 1599, 1711, 1332, 1556,
1811, 1096

Our objective is to develop a sort algorithm that will never require that more than two of these numbers be in memory at one time.

The first step is to read the numbers from the original file in pairs, arrange them in ascending order, and place them alternatively between two files. That is, the first two numbers from the original file are read, arranged in ascending order, and placed in one file. The second pair of numbers is read, arranged in ascending order, and placed in the second file. The result of performing this operation is to produce the following two files:

File C	File D
1214	1417
1796	1906
1047	1215
1814	1516
1108	1599
1604	1711
1332	1096
1556	1811

346 The next step is to merge the first group of two numbers on each file into a single group of four numbers in ascending order on one file. The second group of two numbers on each file is then arranged in ascending order on the second file. The result is

File A	File B
1214	1047
1417	1215
1796	1516
1906	1814
1108	1096
1599	1332
1604	1556
1711	1811

This operation can be performed in much the same manner as the files in Figure 10-8 were merged. No more than one entry from each file need be in memory at one time.

The next step is to process the numbers in groups of four, arrange in ascending order, and place on alternate files. The result is as follows:

File C	File D
1047	1096
1214	1108
1215	1332
1417	1556
1516	1599
1796	1604
1814	1711
1906	1811

We now have two files, each of which is in ascending order. We need only to merge these two files to obtain the final file in ascending order:

1047, 1096, 1108, 1214, 1215, 1332, 1417,
1516, 1556, 1599, 1604, 1711, 1796, 1811,
1814, 1906

In order to perform this operation, four files are required. The sequence of steps using these files is summarized in Table 10-1. For our particular set of data, the sorted list appears on the same file as the original list. This depends upon the number of entries and may appear on File 3. Also, we originally began with 16 points, which is an integral power of 2. Therefore, we always had an equal number of values for each merge. Whenever the number of points is not an integral power of two, merging of sets with an unequal number of points will be encountered, but does not alter the general approach.

Table 10-1 Summary of Steps in External Sort

After Step 1				After Step 3					
Initial		After Step 2				After Step 4			
File	File	File	File	File	File	File	File	File	File
A	B	C	D	A	B	C	D	A	B
1214	—	1214	1417	1214	1047	1047	1096	1047	—
1796	—	<u>1796</u>	<u>1906</u>	1417	1215	1214	1108	1096	—
1417	—	<u>1047</u>	<u>1215</u>	1796	1516	1215	1332	1108	—
1906	—	<u>1814</u>	<u>1516</u>	<u>1906</u>	<u>1814</u>	1417	1556	1214	—
1047	—	<u>1108</u>	<u>1599</u>	<u>1108</u>	<u>1096</u>	1516	1599	1215	—
1814	—	<u>1604</u>	<u>1711</u>	1599	1332	1796	1604	1332	—
1215	—	<u>1332</u>	<u>1096</u>	1604	1556	1814	1711	1417	—
1516	—	1556	1811	1711	1811	1906	1811	1516	—
1604								1556	
1108								1599	
1599								1604	
1711								1711	
1332								1796	
1556								1811	
1811								1814	
1096								1906	

Actually, the subject of external sorting can be treated in far more detail than we have presented in this section. Hopefully, however, this example has shown at least the general approach to the problem. Other strategies have been developed, but the general discussion of various sorting algorithms is beyond the scope of this text.

EXERCISES

- 10-1. Prepare the flowchart for a program that will copy a file and in the process ascertain whether or not it is in sequential order. The name of the record is FILE_RECORD, and is supposed to be in sequential order according to the item SORT_KEY. If any record is found to be out of order, it should be written into a second output file. At the conclusion of the run, the program should print the number of records found to be out of sequence and therefore written into this second file.
- 10-2. Prepare the flowchart for a program to read sales records (SALES_RCD) from an input file named Sales File and prepare an output file named Billing File containing records named BILLING_RCD. The records in Sales File contain the following information:

- 1. Item number
- 2. Item name

348

- 3. Sales price per unit
- 4. Number of units ordered

The records in Billing File are to contain the following information:

- 1. Item name
- 2. Sales price per unit
- 3. Number of units ordered
- 4. Sales amount
- 5. Sales tax (6 percent of sales amount)
- 6. Total amount (sum of sales amount and sales tax)

At the conclusion of the run, the program should print cumulative totals for items 4, 5, and 6 in the records in Billing File.

10-3. Modify the previous problem to include a discount schedule in computing the sales amount. The following discounts apply:

Units ordered	Discount, percent
0 - 11	0
12 - 25	2
26 -100	3
Over 100	5

The discount applies to all items purchased. That is, if 47 units are ordered, the purchaser should be given a discount of 3 percent on all 47 items. An entry for discount amount should be added to the records on the Billing File, and a cumulative total for discount amount should be computed.

10-4. Let us assume that a national television network plans to name a “Quarterback-of-the-Week” during its Sunday pregame show each week of the football season. The selection will be based on statistics read into and evaluated by a computer. The input file, Quarterbacks, will contain punched-card records, PLAYER_INFO, which will have the following information:

- 1. Name of player
- 2. Attempted passes
- 3. Completed passes
- 4. Interceptions
- 5. Number of carries
- 6. Rushing yardage
- 7. Passing yardage

The program must be written to read a card and then compute the percentage of completed passes, the average yardage per pass, the average yardage per carry, and net yardage. The score for each quarterback will be computed according to the following formula:

$$\text{Score} = \frac{\text{pass completion percentage}}{20} + \frac{\text{avg yd/pass}}{5} + \frac{\text{avg yd/rush}}{5} - \text{interceptions}$$

If a player's total score is greater than a previous player's total score, his name will be moved into an item called QUARTERBACK_OF_THE_WEEK. Each player's computed record will be written as hard copy after it has been processed. Finally, the player's name remaining in QUARTERBACK_OF_THE_WEEK will be displayed with an appropriate message. Prepare the flowchart for this program.

10-5. Our company maintains an Inventory File whose records contain the following information on each item:

1. Item stock number
2. Item description
3. Quantity in inventory
4. Quantity backordered
5. Quantity on order
6. Sales price of item
7. Purchase price on last shipment received
8. Quantity in inventory at which another order is to be placed
9. Quantity to be ordered
10. Source of item (name and address)

This file is maintained in ascending order of the item stock number. Prepare the flowchart for a program to add new stock numbers to the Inventory File.

10-6. For the Inventory File described in the above exercise, prepare the flowchart for a program to delete items from the Inventory File. In making deletions, our company maintains a file called the Deleted Inventory File that contains items 1, 2, 6, and 10 of the Inventory File. This file is also in ascending order of stock numbers, and may contain duplicate entries for a given stock number. If any units of the item to be deleted are either in stock or on order, the record in the Inventory File is copied into the Disposal File. The stock numbers of items to be deleted are read from cards.

10-7. Refer to the Inventory File in Exercise 10-5. For items received, cards are prepared containing the following information:

1. Item stock number
2. Quantity received
3. Price

These are sorted to form the Items Received File, which may contain duplicate entries for a given stock number. Prepare the flowchart for the program to update the Inventory File. The program should take an appropriate action upon

detection of a stock number in the Items Received File that does not exist in the Inventory File.

10-8. Refer to the Inventory File in Exercise 10-5. Prepare the flowchart for a program that places orders (that is, prepares an Order File) for items whose quantity in inventory has fallen below the quantity at which another order is to be placed. If the quantity in inventory plus the quantity on order is below this limit, another order should be placed and a message printed to the effect that an order has been placed for an item for which a shipment is already on order. From these messages, the purchasing agent can investigate the situation for potential problems.

10-9. Refer to the Inventory File in Exercise 10-5. Prepare the flowchart for a program to update entries 2, 6, 8, 9, or 10 in this file. For any given item, changes may be made in one or all of these entries.

10-10. Prepare the flowchart for a program that will make additions, deletions, or changes to the records in the Class Section File described in Section 10.2. The sequential order of this file should be maintained.

10-11. This problem is concerned with the printing of paychecks for employees paid on an hourly basis. For each employee, the respective supervisor prepares a time sheet specifying the following information for the past two-week period:

1. The ID number of the employee
2. Regular hours worked
3. Overtime hours worked
4. Hours charged to vacation
5. Hours charged to sick leave

The employee is paid at the base rate for hours in categories 2, 4, and 5, and at 1.5 times the base rate for overtime. The data on these time sheets are punched onto cards, which are then read and the information written onto magnetic tape, and finally the records are sorted according to employee ID number to form the Time File.

In processing the data on the Time File, information must be obtained from the Master Record File, and some entries in this file must be updated. Each record in this file contains the following entries:

1. Employee's ID number
2. Employee's name
3. Employee's address
4. Date on which employment began
5. Number of dependents
6. Base hourly pay rate
7. Total regular hours worked to date this year
8. Total overtime hours worked to date this year
9. Total hours charged as vacation to date this year
10. Total hours charged as sick leave to date this year

11. Income tax rate, percent of total pay
12. Cumulative income tax withheld to date this year
13. Cumulative FICA withheld to date this year
14. Deductions
 - a. Insurance
 - b. Retirement
 - c. United Givers
15. Cumulative deductions to date this year
 - a. Insurance
 - b. Retirement
 - c. United Givers

As an integral part of the processing run, items 7, 8, 9, 10, 12, 13, and 15 should be updated.

In computing the FICA withholding, the contribution is withheld at a rate of FICA_PCT in percent until the total withheld to date reaches FICA_MAX.

This program should produce a file called Paycheck File, each record containing the following information:

1. Employee's ID number
2. Employee's name
3. Regular hours (including vacation and sick leave)
4. Overtime hours
5. Total pay
6. Withholdings
 - a. Income tax
 - b. FICA
 - c. Insurance
 - d. Retirement
 - e. United Givers
7. Net pay

The program should also check for the following conditions and print messages accordingly:

1. A record in the Time File with no corresponding entry in the Master Record File
2. A record in the Master Record File with no corresponding entry in the Time File

Only the employee ID number need be printed along with information designating the condition that generated the message.

At the conclusion of the run, the program should print the following summary:

1. Number of paychecks issued
2. Cumulative totals for all employees for this pay period
 - a. Total pay

- b. Income tax withheld
- c. FICA withheld
- d. Contributions withheld
 - (1) Insurance
 - (2) Retirement
 - (3) United Givers
- e. Net pay

10-12. At the end of the year, the final version of the Master Record File is retained for possible future reference. To create a Master Record File for the coming year, the old Master Record File is copied with all cumulative totals set equal to zero. Prepare the flowchart for this program.

10-13. Periodically, it is necessary to make a number of changes in the data in the Master Record File described in Exercise 10-11. Prepare the flowchart for a program that will perform the following:

1. Selectively change the entries for items 2, 3, 5, 6, 11, and 14 in the records in the Master Record File.
2. Delete records from the Master Record File, entering them into the Inactive Personnel File (which is maintained in ascending order of employee ID numbers).
3. Add records to the Master Record File. In so doing, the cumulative totals are set equal to zero.

10-14. The program in Exercise 10-11 printed the employee ID number for all entries in the Time File for which no corresponding entry was found in the Master Record File. Many installations find it more convenient to handle these exceptions by hand rather than perform another processing run. One possible cause for errors of this type is that an error was made in entering the employee ID number in the Time File. This can usually be corrected readily and a paycheck issued, but the cumulative totals in the Master Record File must be updated. Add this operation to the program described in the previous exercise.

10-15. The objective of this problem is to prepare a system chart for the term grade reporting system for a large university. As a result of the registration system, the following two files are created for use in the grade reporting system:

1. The Student Class File contains the necessary personal data on the student along with the designation of the courses he is taking.
2. The Class Records File contains the necessary data describing each section of each course offered this term along with the ID numbers and names (in alphabetical order) of the students enrolled.

For identification purposes, each student is given an ID number, and each section of each course is given a unique "line number."

From the Class Records File, a grade sheet is printed for each course. This sheet identifies the course and lists the students' ID numbers and names in

alphabetical order. The instructor then enters the grade for each student. Upon return, these forms are read by an optical scanner which detects only the line number of the course and the grade entered (but *not* the student's name or ID number). Let us avoid the sticky question of incorrect rolls by assuming that the instructors are not permitted to add or delete names on the grade sheet.

Draw the chart for a system that accepts these grade sheets and prints grade reports to the "folks back home." Be sure to include error checking at each stage of the process.

10-16. Our company has completely automated its sales accounting system. After making a sale, the salesman sends to headquarters a form containing the following information:

1. Numerical designation of order number
2. Customer's name and address
3. Salesman's ID number
4. The stock number and quantity for each item ordered (up to a maximum of twenty items)

This information is then punched onto cards to create an Order File.

Our company maintains another file known as the Inventory File that contains the following information on each item in inventory:

1. Stock number for the item
2. Name of the item
3. Sales price
4. Quantity in inventory
5. Salesman's commission (percent)

This file is in ascending order of the stock numbers.

Our company also maintains a file known as the Salesmen File that contains the following information on each salesman:

1. Salesman's ID number
2. Cumulative sales, month to date
3. Cumulative commission, month to date

The records in this file are in sequential order according to the salesman's ID number.

Prepare the system chart for accomplishing the following:

1. Print a complete invoice showing items to be shipped, sales price of each item, and total price of all items on the invoice.
2. Update the inventory file for items shipped.
3. If there is not a sufficient quantity of the respective item in inventory, a partial shipment will be made and the extra items "backordered", which necessitates the creation of a Backorder File. The price on the customer's invoice should reflect only the items shipped.
4. The Salesmen File should be updated.

11

NUMERICAL APPLICATIONS

In the scientific community, the use of the digital computer to solve numerical problems has revolutionized many fields. In fact, it has spurred the development of a new area of mathematics, namely, applied mathematics, within whose realm the subject of numerical techniques lies.

In this chapter, we shall treat several subjects pertaining to numerical techniques.

11.1. FINITE ARITHMETIC

Most of the theories and principles taught in the typical course in mathematics assume that infinite precision is available. Consequently, we would say that

$$3 \times (10/3) = 10$$

If, however, the number of digits used to represent a number is restricted to any finite value, the above relationship no longer holds. For example, with three digits of accuracy,

$$3 \times (10/3) = 3 \times 3.33 = 9.99$$

which is not quite 10. Using eight digits, the answer is 9.9999999, which is closer to but still not equal to 10.

On the other hand, the relationship

$$(3 \times 10)/3 = 10$$

holds as long as one or more digits of accuracy is available. That is,

$$(3 \times 10)/3 = 30/3 = 10$$

Combining this result with that in the previous paragraph, we conclude that

$$3 \times (10/3) \neq (3 \times 10)/3$$

for finite arithmetic. In other words, laws of associativity do not hold in general for finite arithmetic.

It should be pointed out that this phenomenon is strictly due to the fact that we are using finite arithmetic; that is, we are restricted to a finite number of digits for representing quantities. Under these conditions, there are many values, for example, π , $\sqrt{2}$, $1/7$, that cannot be represented exactly. This is apart from the problem pointed out in Chapter 6, namely, that the binary floating point notation cannot exactly represent most decimal numbers.

On most computers, the culprit is floating-point arithmetic. In effect, numbers are represented in the scientific or exponential format:

$$\pm 0.d_1d_2 \dots d_s \times 10^e$$

where $d_1, d_2, \dots, d_s$ are the digits and e is an integer exponent. While most computers use either base-2, base-8, or base-16 representations, their accuracy is typically limited to the equivalent of about eight decimal digits (not counting leading or trailing zeros).

To illustrate what can happen, suppose we consider the addition process when we are restricted to four digits. Specifically, suppose we add the following numbers:

$$\begin{array}{r} 6.954 \\ -6.953 \\ 0.002 \end{array}$$

If we add the top two numbers first, we obtain 0.001000 (the customary floating point arithmetic unit will add zeros to the right to give a four-digit representation exclusive of leading zeros). Then adding 0.0022 gives the result 0.003200.

If we change the order in which the operations are performed, a different answer is obtained. Adding 0.0022 to -6.953 gives, to four digits, -6.951 , and then adding 6.954 gives 0.003000, which is not equal to 0.003200 as determined earlier. Thus, the order in which operations are performed is important.

In effect, we have again shown that the associative property does not hold in general for finite arithmetic. That is,

$$(A + B) + C \neq A + (B + C)$$

for finite arithmetic.

Another feature of finite arithmetic is that there exists a smallest positive number. In ordinary mathematics (unlimited number of digits), there is no

356 smallest number. Should anyone propose a candidate for such a number, you need only divide it by two to obtain an even smaller one.

In floating-point representations, limits are imposed on the size of the exponent, a typical set of limits being

$$-76 \leq e \leq 76$$

For this case, the smallest positive number (again using four digits) is

$$0.1000 \times 10^{-76}$$

Floating-point representations generally insist that the number be normalized; that is, first digit following decimal point must be nonzero. If we divide 0.1000×10^{-76} by 2, we obtain

$$0.0500 \times 10^{-76}$$

or after normalization

$$0.5000 \times 10^{-77}$$

Now the exponent is too large negatively, and an *underflow* is said to occur. On many machines, the result is that the number is set equal to zero. Therefore, 0.1000×10^{-76} is the smallest positive number.

Conventional mathematicians are uncomfortable in this region. The associative property does not hold; there is a smallest positive number. Nor does the distributive property hold in general. To the programmer, this means that he must be careful.

As another example of problems associated with finite arithmetic, consider computation of the value of the following polynomial for $x = 3.7$:

$$x^3 - 4.2x^2 + 0.4x + 5.2$$

Using infinite precision, the answer is

$$\begin{array}{r} 50.653 \\ - 57.498 \\ 1.480 \\ 5.200 \\ \hline - 0.165 \end{array}$$

Limiting ourselves to three digits accuracy and working from the top, we obtain

$$\begin{array}{r} 50.7 \\ - 57.5 \\ \hline - 6.80 \\ + 1.48 \\ \hline - 5.32 \\ + 5.20 \\ \hline - 0.120 \end{array}$$

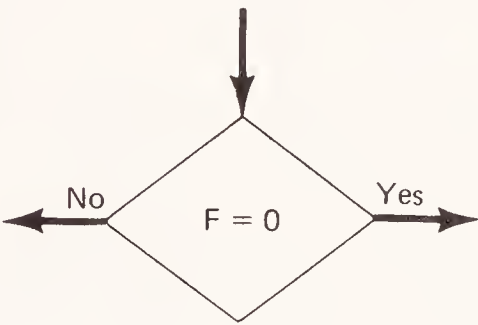
which is about 25 percent lower than the previous answer.
As pointed out in Exercise 2-5, a more efficient manner for evaluating polynomials is as follows:

$$((x - 4.2)x + 0.4)x + 5.2$$

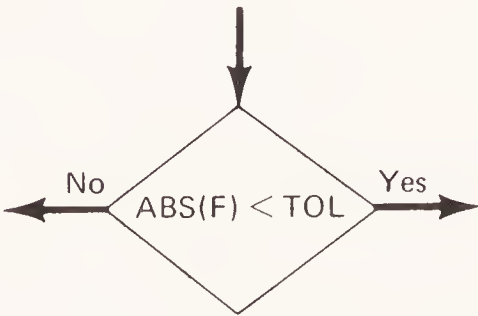
Again using three digits accuracy and computing, we obtain

$$\begin{aligned} 3.70 - 4.20 &= -0.500 \\ (-0.500) \cdot (3.7) &= -1.85 \\ (-1.85 + 0.4) &= -1.45 \\ (-1.45) (3.7) &= -5.37 \\ -5.37 + 5.2 &= -0.170 \end{aligned}$$

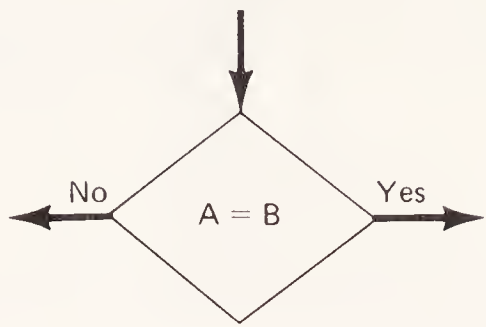
This is much closer to the “exact” answer.
Finally, we must observe that exact floating-point comparisons are meaningless. In all such comparisons, the numbers are said to be equal if they are within a certain tolerance of each other. For example, instead of



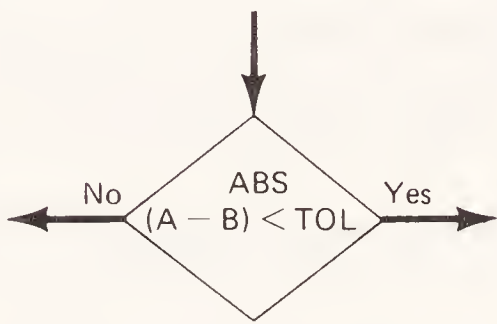
we should use



358 when TOL is a small positive number. Similarly,



should be replaced by



11.2. USE AND EVALUATION OF SERIES

Before the end of the first calculus course, students generally learn how to represent functions such as $\sin x$ and e^x by use of a series such as Taylor's or Maclaurin's. For example, using Maclaurin's series gives the following representations for $\sin x$ and e^x :

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!} + \dots$$
$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^i}{i!} + \dots$$

Since the use of the customary table would require large amounts of storage space, it is by such series, although not necessarily the above series, that computers evaluate many functions.

In programming the evaluation of series such as those above, the natural tendency is to use a loop to sum the series in a straightforward manner. Let's assume that we have a function subprogram FACT that will compute factorials (see Exercise 3-18), that is, $i! = \text{FACT}(i)$. Then one approach to using the above series to compute $\sin x$ is given in Figure 11-1.

This is not, however, a very efficient approach. First the factorials are recomputed at each iteration. Second, the exponentials $(-1)^i$ and x^{2i-1} are also recomputed. A more efficient approach would be to use the ratio of term i to term $(i-1)$ to compute the current term from the previous one. For example,

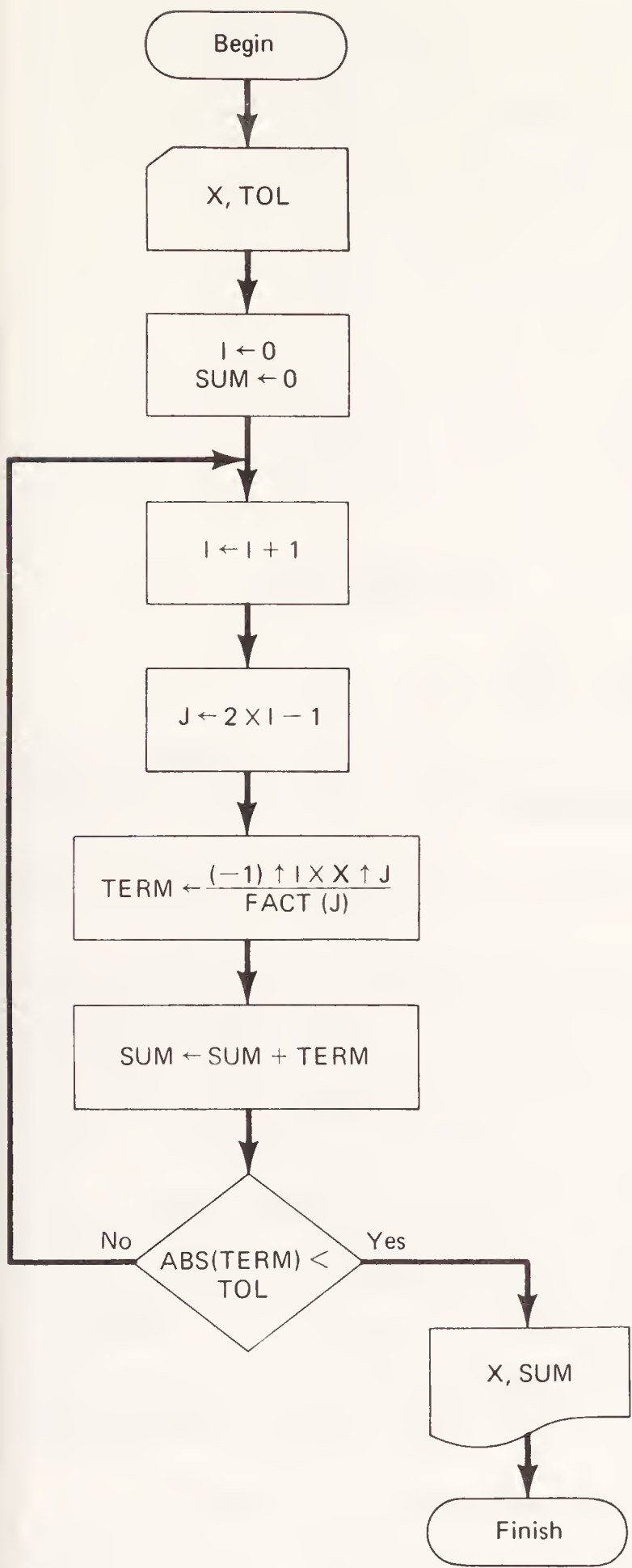


Figure 11-1 Computation of $\sin x$ via the Maclaurin series

$$\text{term } i = - \frac{(-1)^i x^{2i-1}}{(2i-1)!}$$

$$\text{term } (i-1) = - \frac{(-1)^{i-1} x^{2i-3}}{(2i-3)!}$$

360 The ratio is computed as follows:

$$\begin{aligned} \frac{\text{term } i}{\text{term } (i-1)} &= \frac{-\frac{(-1)^i x^{2i-1}}{(2i-1)!}}{-\frac{(-1)^{i-1} x^{2i-3}}{(2i-3)!}} \\ &= \frac{(-1)^i}{(-1)^{i-1}} \cdot \frac{x^{2i-1}}{x^{2i-3}} \cdot \frac{(2i-3)!}{(2i-1)!} \\ &= (-1) \cdot x^2 \cdot \frac{1}{(2i-2)(2i-1)} \\ &= -\frac{x^2}{(2i-2)(2i-1)} \end{aligned}$$

The first term is simply x , and each successive term can be computed using this ratio. The flowchart in Figure 11-2 embodies this logic and is considerably more efficient than the one in Figure 11-1.

The same approach can be used in preparing a flowchart to compute e^x . This is left to the exercises.

While it can be established that these series converge to the correct answer for all values of x , the proof is for the case of unlimited accuracy. When using finite arithmetic, they converge, but not necessarily to the correct answer. Stegun and Abramowitz† use the evaluation of $e^{-5.5}$ to illustrate clearly what can happen. Using a machine that carries five digits, the terms for the evaluation of $e^{-5.5}$ via Maclaurin's series are given in Table 11-1. The stopping criterion is whenever the value of a term in the series is too small to change the value of the sum, which happens after the twenty-fifth term. Unfortunately, the final answer of 0.0034544 does not agree especially well with the true answer of 0.00408677. In fact, it is off by about 15 percent, an intolerable error. The problem is the use of finite arithmetic to sum a series in which small differences between large numbers must be computed.

In this case, a better approach would be to compute $e^{-5.5}$ as follows:

$$e^{-5.5} = \frac{1}{e^{5.5}}$$

Computing in this manner and again using five-digit accuracy for computing $e^{5.5}$, the answer is 0.0040868, which is negligibly in error.

In effect, the above example illustrates that series such as Taylor's or Maclaurin's are not always sufficient for evaluation of transcendental functions. Care must be exercised to prevent the peculiarities of finite arithmetic from causing the result to be in significant error.

†Irene A. Stegun and Milton Abramowitz, "Pitfalls in Computation," *J. Soc. Indust. Appl. Math.*, 4 (1956), 207-219.

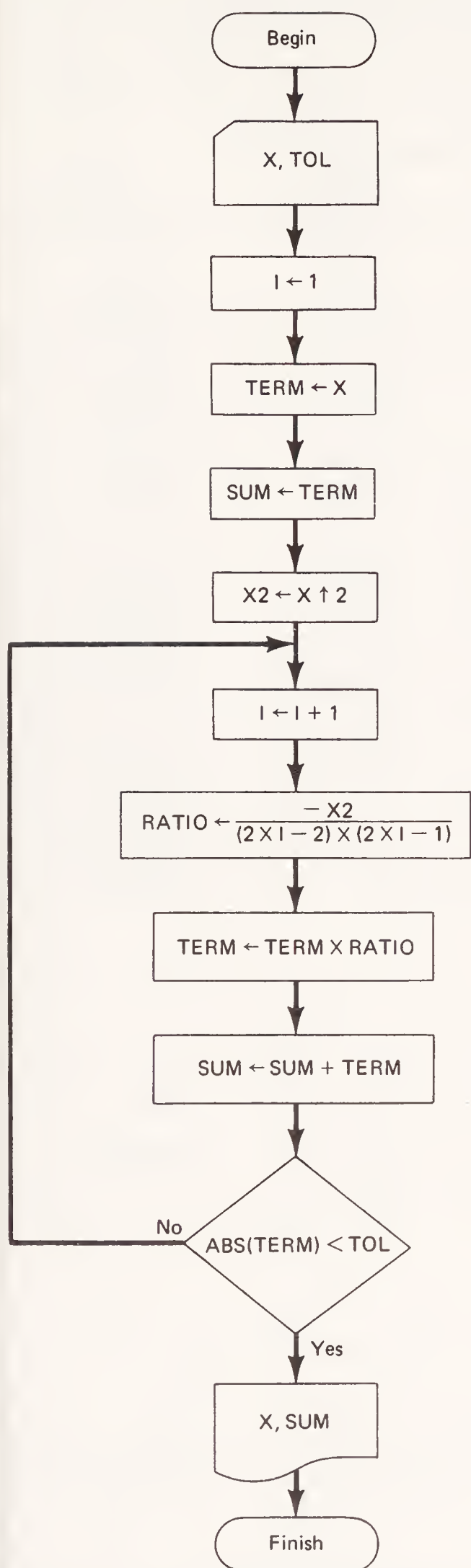


Figure 11-2 Improved algorithm for computing $\sin x$ via the Maclaurin series

Table 11-1 Evaluation of e^x Using Finite Arithmetic (Five Digits)

Index	$e^{-5.5}$		$e^{5.5}$	
	Term	Sum	Term	Sum
0	1.0000	1.0000	1.0000	1.0000
1	− 5.5000	− 4.5000	5.5000	6.5000
2	15.125	10.625	15.125	21.625
3	−27.729	−17.104	27.729	49.354
4	38.127	21.023	38.127	87.481
5	−41.940	−20.917	41.940	129.42
6	38.445	17.528	38.445	167.86
7	−30.207	−12.679	30.207	198.07
8	20.767	8.0880	20.767	218.84
9	−12.691	− 4.6030	12.691	231.53
10	6.9800	2.3770	6.9800	238.51
11	3.4900	− 1.1130	3.4900	242.00
12	1.5996	0.48660	1.5996	243.60
13	− 0.67676	− 0.19016	0.67676	244.28
14	0.26587	0.075710	0.26587	244.55
15	− 0.097487	− 0.021777	0.097487	244.65
16	0.033511	0.011734	0.033511	244.68
17	− 0.010842	0.00089200	0.010842	244.69
18	0.0033129	0.0042049	0.0033129	244.69
19	− 0.00095898	0.0032459	$e^{5.5} =$	244.69
20	0.00026372	0.0035096		
21	− 0.00006907	0.0034405		
22	0.00001727	0.0034578		
23	− 0.00000413	0.0034537		
24	0.00000095	0.0034546		
25	− 0.00000021	0.0034544	$e^{-5.5} =$	$\frac{1}{e^{5.5}}$
26	0.00000004	0.0034544	$= \frac{1}{244.69} =$	0.0040868
	$e^{-5.5} =$	0.0034544		

11.3. SOLUTION OF NONLINEAR ALGEBRAIC EQUATIONS

The solution of nonlinear algebraic equations is necessary in the solution of a number of problems. One example is the solution of equations such as

$$x^4 + 10.62x^3 + 14.76x^2 + 15.21x + 5.58 = 0$$

Since this is a fourth-order equation, we know that there are four roots, but we do not know where any of them lie. In some cases, the physical formulation of

the problem suggests where one or more of the roots should lie; indeed, in many cases we are interested in only one of the roots.

Figure 11-3 gives a plot of the left hand side of the above equation:

$$f(x) = x^4 + 10.62x^3 + 14.76x^2 + 15.21x + 5.58$$

From this graph, it is obvious that one of the roots lies between $x = 0$ and $x = -1$.

To determine this root accurately, a process known as *bisection*, or *interval halving*, can be used. This technique is applicable whenever it is known that a root lies within some fixed interval.

To illustrate the approach, the $(-1,0)$ interval of Figure 11-3 is enlarged in Figure 11-4. The first step in the interval halving technique is to compute the value of $f(x)$ at the midpoint of the interval, that is, $f(-0.5)$. For the curve illustrated in Figure 11-4, $f(-0.5)$ is positive. Although the small value for $f(-0.5)$ indicates that the root is close to -0.5 , this is of no interest. That is, we want to know only if $f(-0.5)$ is positive or negative.

For the curve drawn in Figure 11-4, if the functional value is positive, the root must lie to the left. That is, since $f(-0.5)$ is positive, the root must lie between -1 and -0.5 . If $f(-0.5)$ had been negative, the root must then lie between -0.5 and 0 . In either case, we have cut in half the interval over which the root is known to lie.

The next step is to repeat this procedure for the new interval, that is, $(-1, -0.5)$. Evaluation of $f(x)$ at the midpoint, that is, $f(-0.75)$, gives a negative value.

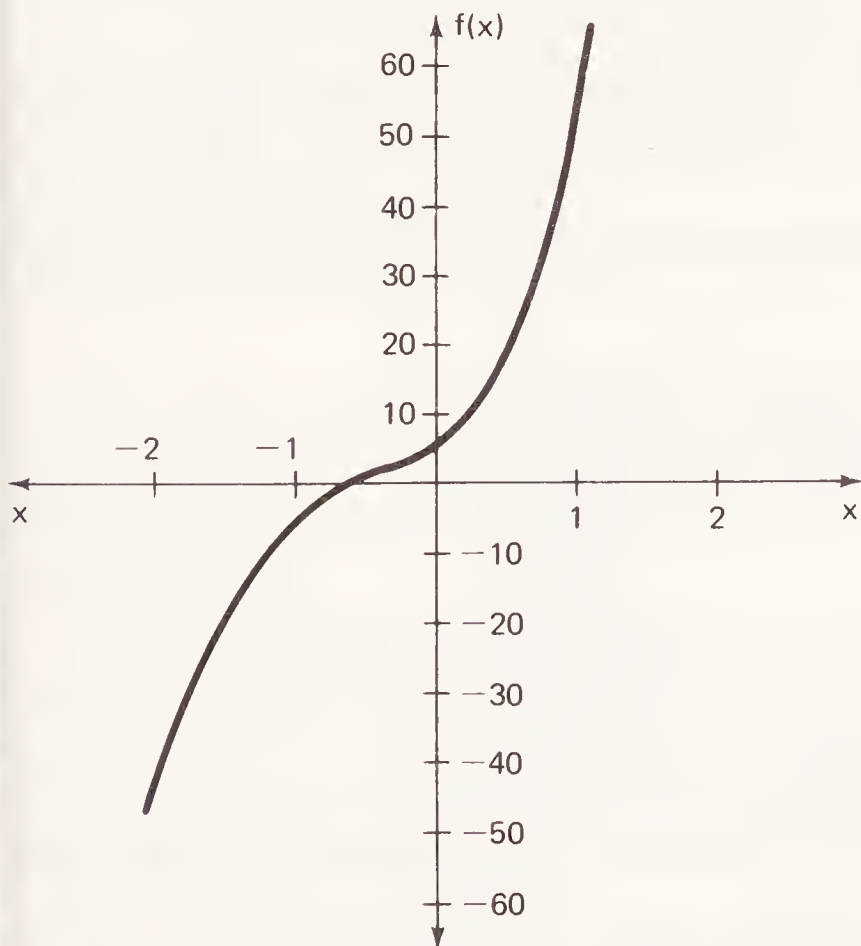


Figure 11-3 Plot of $f(x)$

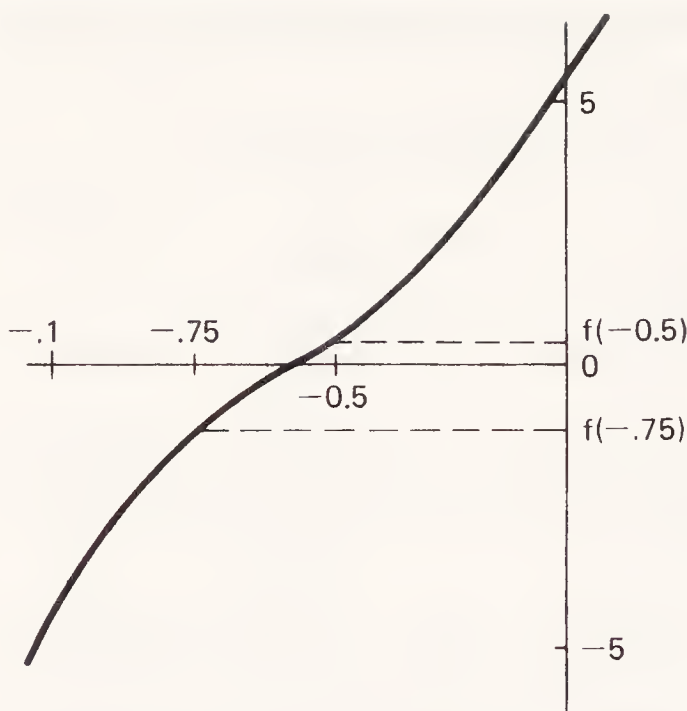


Figure 11-4 Interval halving

Therefore, the root must lie within the interval $(-0.75, -0.5)$. Again, we have cut the interval in half.

After two “iterations,” that is, function evaluation and updating of the interval of uncertainty, the resulting interval is now $1/4$ of the original interval. After three iterations, the interval will be $1/8$, after four, the interval will be $1/16$, etc. After twenty iterations, the interval will be reduced to $1/2^{20}$, or about one-millionth, of the original interval. In other words, we shall know the root to within 0.000001 of its true value. In general, this is more than adequate, but if not, more iterations can be made.

The flowchart for this algorithm is given in Figure 11-5. An iteration counter is used to cause the program to loop 20 times. Upon completion of the iterations, the answer printed is the midpoint of the final interval.

In many root-solving problems, the equations do not appear as an explicit algebraic equation of high order. An example of such a situation comes from the field of business.

Suppose a firm invests \$75,000 and obtains the following returns over the next five years:

Year	Return
1	20,000
2	30,000
3	35,000
4	40,000
5	50,000
Total	\$175,000

Since the total return exceeds the investment by \$100,000, it looks very attractive.

The total return does not, however, take into account the time value of money. Suppose we are operating in an environment in which the current

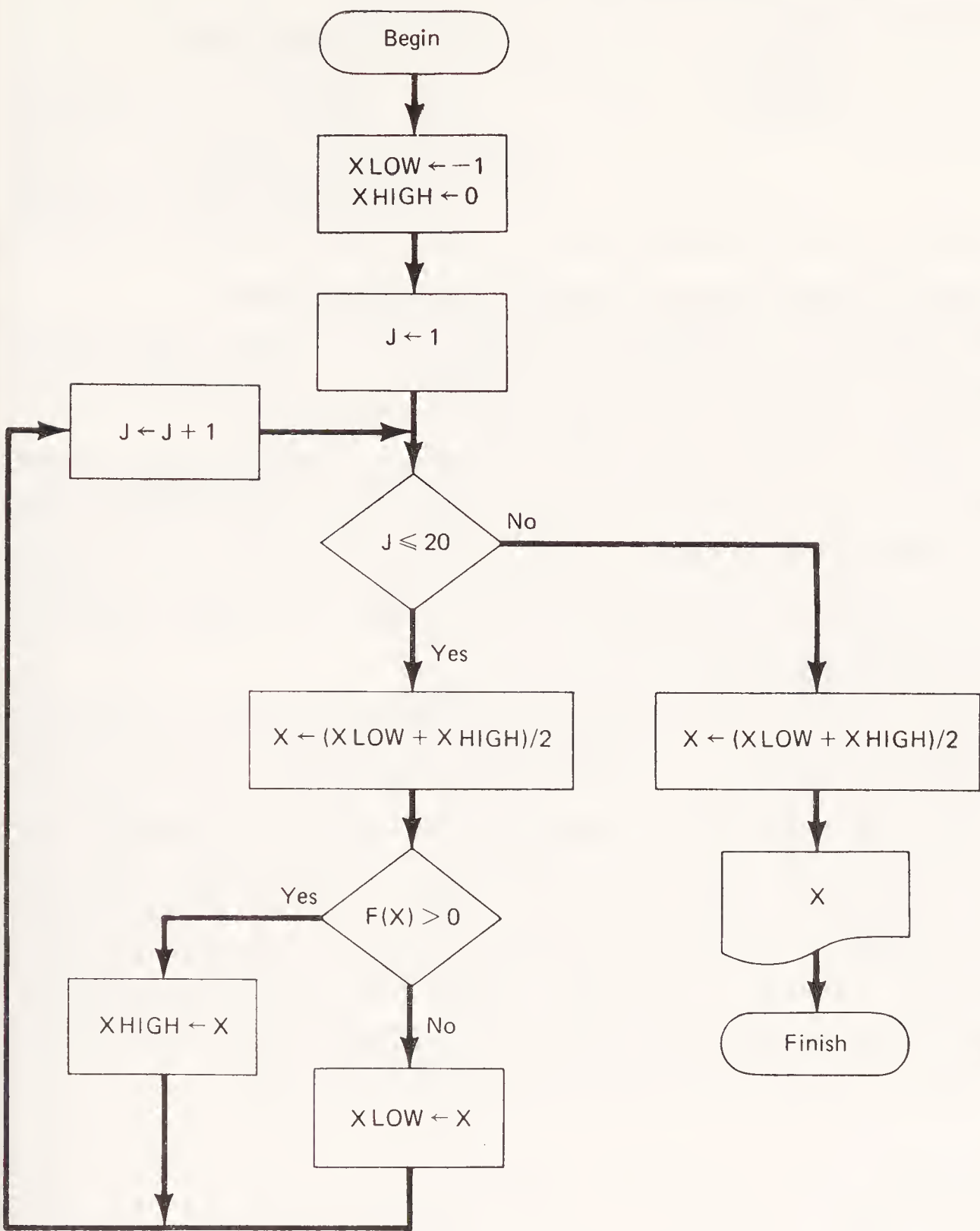


Figure 11-5 Flowchart for the bisection algorithm

interest rate is 10 percent. To obtain \$20,000 after one year, we would have to invest \$20,000/1.1, or \$18,182. That is, \$20,000 one year from now is equal to \$18,182 today. Similarly, \$30,000 two years from now is worth \$30,000/(1.1)² = \$24,794 today.

The same analysis can be applied to the remaining returns, which permits the *net present value* (NPV) for the investment to be computed as follows:

Year	Return	Present value	
1	20,000	20,000/1.1 =	18,182
2	30,000	30,000/(1.1) ² =	24,794
3	35,000	35,000/(1.1) ³ =	26,296
4	40,000	40,000/(1.1) ⁴ =	27,321

Year	Return	Present value
5	50,000	$50,000/(1.1)^3 = 31,047$
		Present value of returns = 127,640
		Less investment = - 75,000
		Net present value = \$ 52,640

In effect, the proposed investment would be worth \$52,640 today.

Another approach to analyzing such investments is to ask at what interest rate would the net present value of the investment be zero? In business circles, this interest rate is referred to as the *discounted cash flow* (DCF).

This is in effect a root-solving problem. The graph of the net present value vs. the interest rate is given in Figure 11-6. Although not necessarily the case, the discounted cash flow generally lies between 0 and 100 percent. At an interest rate of zero, the NPV is simply the total return less the investment, or \$100,000 for our case. At 100 percent, the NPV can be computed as -\$59,063. Therefore, the DCF does lie between 0 and 100 percent.

To compute the DCF, we can again use the bisection method, as illustrated by the flowchart in Figure 11-7. The returns and investments are read, the initial upper and lower limits assigned, and the interval halving procedure begun. For each iteration, the interest rate at which the NPV is to be computed is the average of the current upper and lower limits. The net present value corresponding to this interest rate is computed in the inner loop. Upon exit from this loop, the upper or lower limit is updated. In this case, if the NPV is positive, the lower limit is updated; if the NPV is negative, the upper limit is updated. Note that the slope of the curve in Figure 11-6 is opposite to that of the curve in Figure 11-4, and consequently the limits are updated differently.

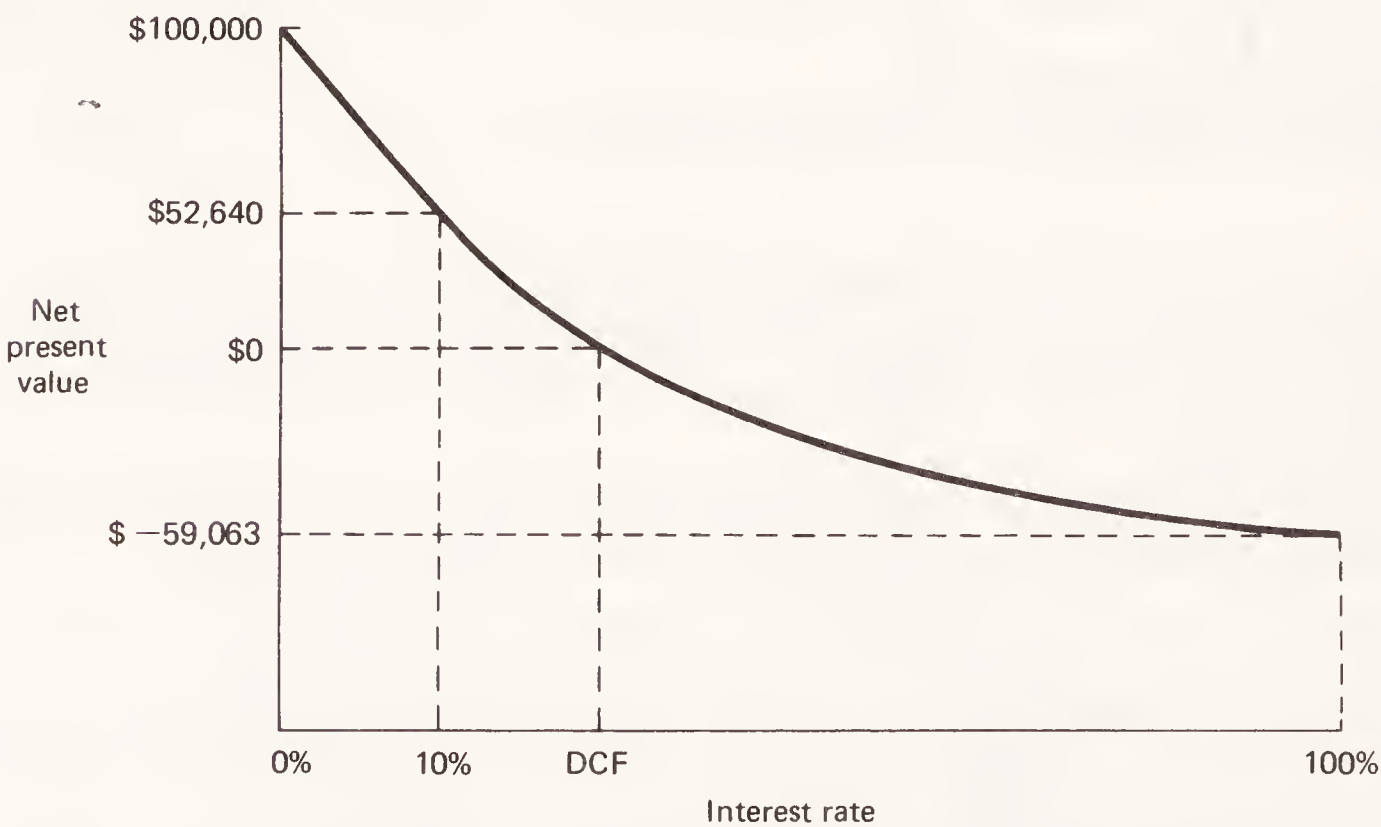


Figure 11-6 The NPV versus interest rate

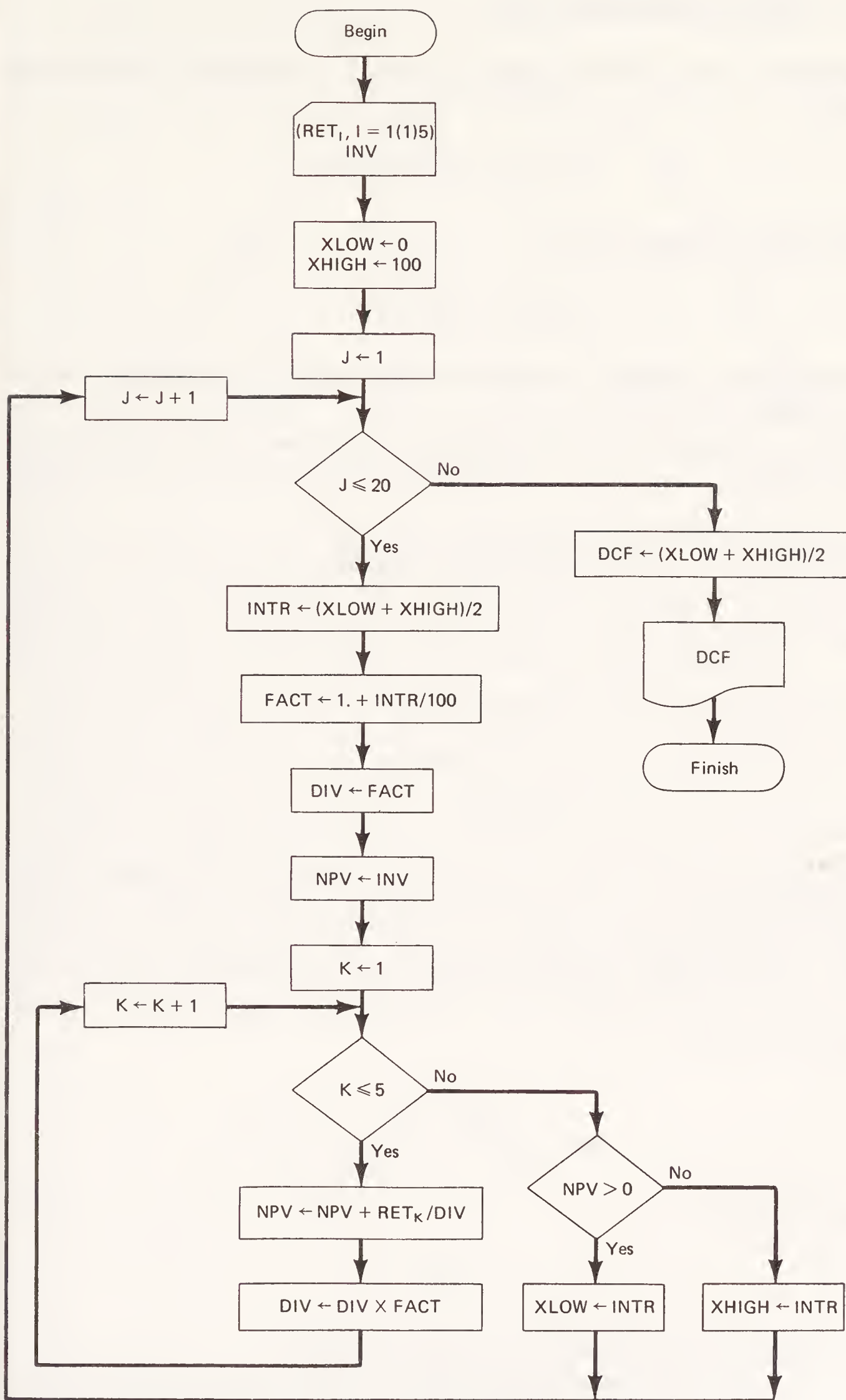


Figure 11-7 Flowchart for computing the DCF

11.4. THE GOLDEN SECTION

A manufacturer can produce x units of an item at a cost given by the following equation:

$$\text{Cost} = C = 15 + 0.001x^2$$

To sell x units, the price must be

$$\text{Price} = P = 35 - 0.1x$$

The manufacturer would like to know how many units he must produce in order to maximize his total return.

This is an example of a one-dimensional optimization problem which can be stated as follows:

$$\text{Max}_x \{x \cdot (P - C)\}$$

where

$$P = 35 - 0.1x$$

$$C = 15 + 0.001x^2$$

$$x > 0$$

The quantity $x \cdot (P - C)$ to be optimized is frequently referred to as the *criterion function*.

Since the manufacturer makes no profit if he sells no goods, x must be positive. Similarly, he makes no profit if the selling price is less than the manufacturing cost. The value of x corresponding to this condition can be computed as follows:

$$\begin{aligned} P &= C \\ 35 - 0.1x &= 15 + 0.001x^2 \end{aligned}$$

Solving for x gives

$$x = 100$$

Thus, the optimum must be within the interval $(0, 100)$.

One approach to determine the optimum is to plot the criterion function versus the independent variable as illustrated in Figure 11-8. From this plot, we can see that the optimum is around 55.

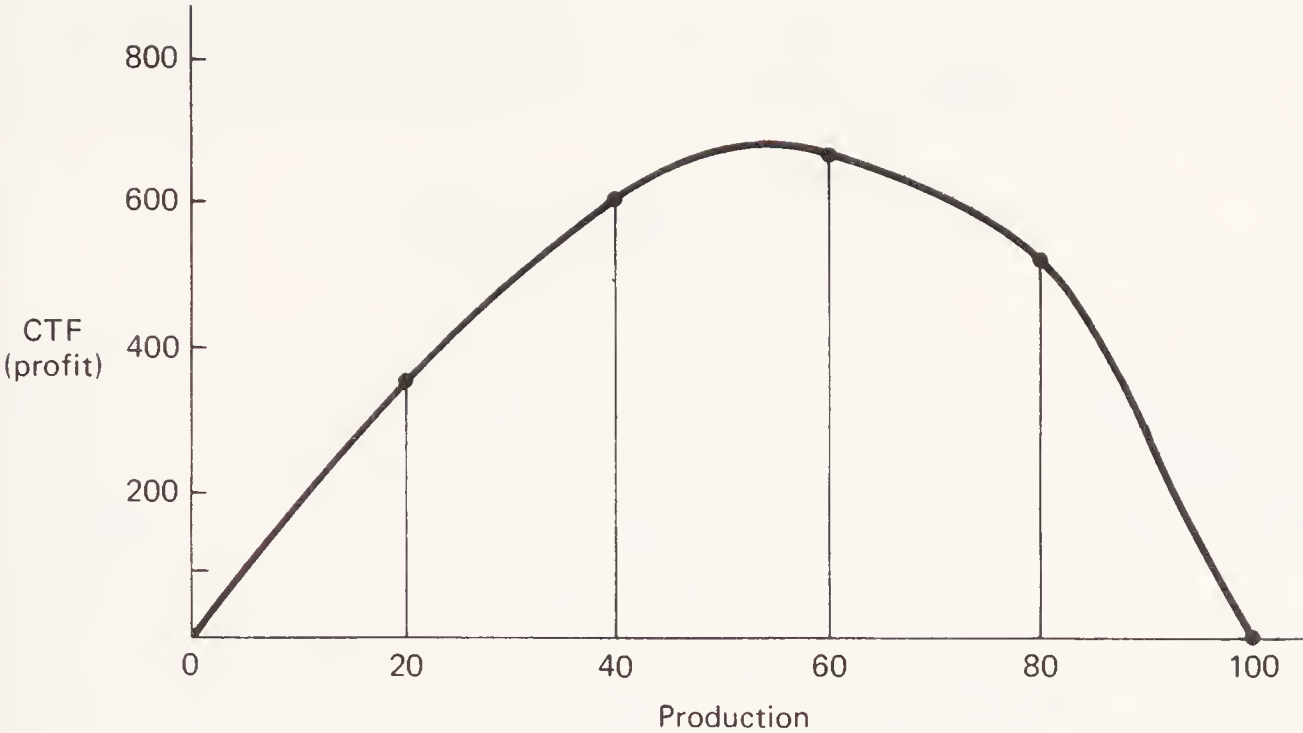


Figure 11-8 Plot of profit versus production

To do this on a computer, we can use a technique known as the *golden section*. This method is applicable only when the interval over which the optimum lies is known and when it is known that there is only one peak over this region; that is, the function is unimodal. If the curve is multimodal, such as the one in Figure 11-9, the golden section will not function properly.

Like the interval halving method, the golden section successively reduces the interval over which the optimum is known to lie. To accomplish this, the criterion function will be evaluated at various points within the interval.

When looking for the optimum, a single evaluation within the interval is of little value. As illustrated in Figure 11-10a, the optimum may lie on either side of this point. But if two points are evaluated within the interval such as in Figures 11-10b or 11-10c, a portion of the interval can be eliminated. In Figure 11-10b $f(x_2) > f(x_1)$; consequently, the optimum must lie within the interval (x_1, x_H) and the interval (x_L, x_1) can be eliminated. In Figure 11-10c, $f(x_1) >$

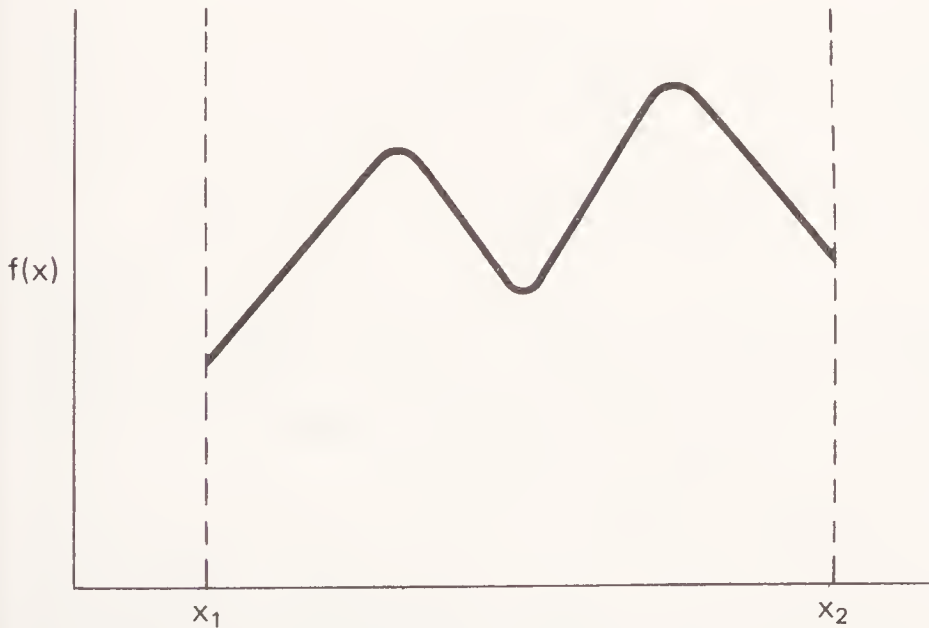
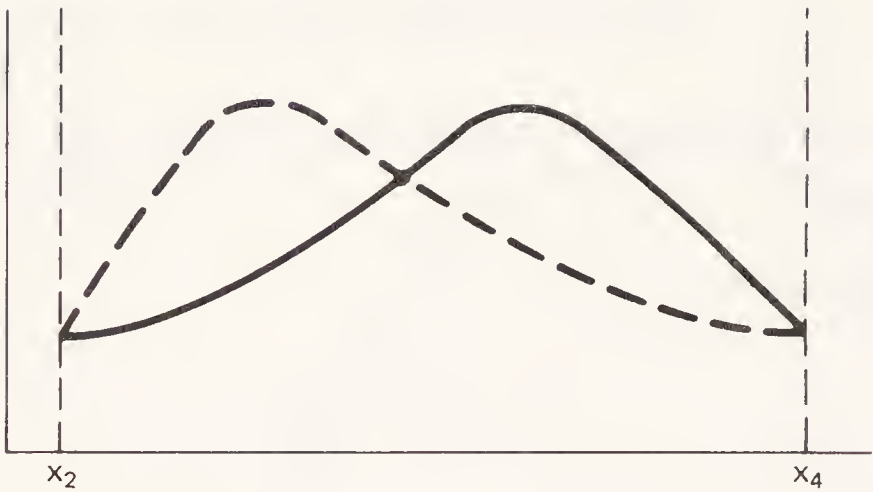
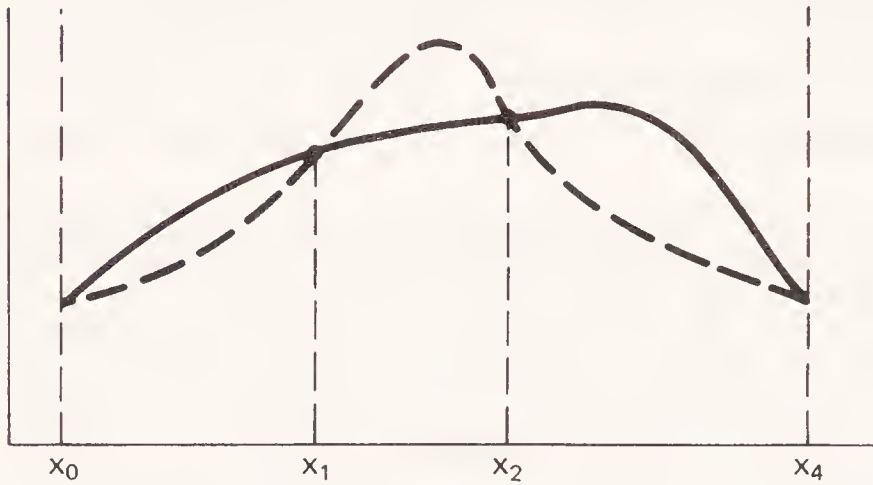


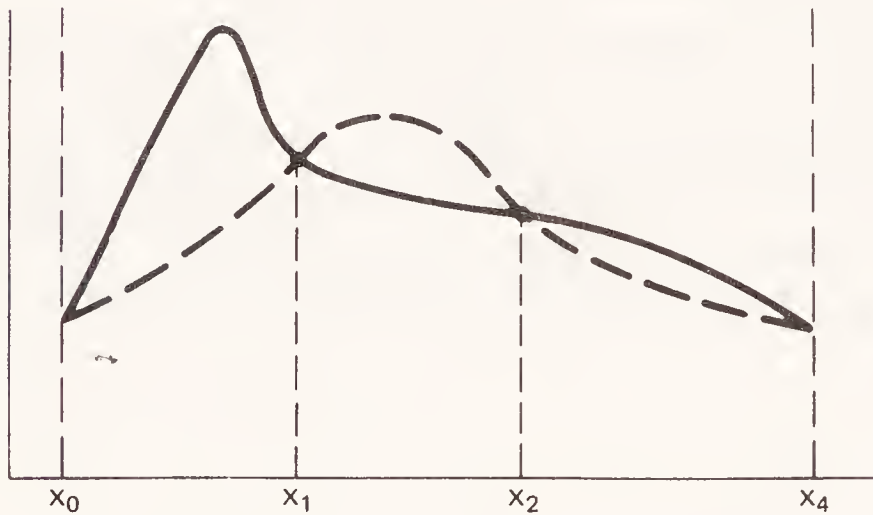
Figure 11-9 Example of a multimodal function over the interval (x_1, x_2)



a. One point



b. Two points with $f(x_2) > f(x_1)$



c. Two points with $f(x_1) > f(x_2)$

Figure 11-10 Placement of points within the interval

$f(x_2)$; consequently, the optimum must lie within the interval (x_L, x_2) and the interval (x_2, x_H) can be eliminated. Should $f(x_2) = f(x_1)$, the optimum must lie over the interval (x_1, x_2) . However, the chances of this happening in a practical problem are so remote that it will not even be considered.

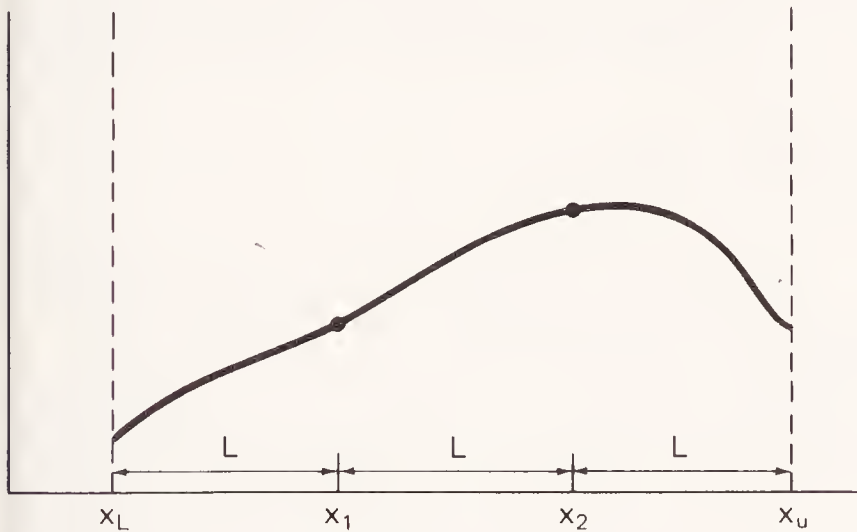
The question now before us is where should we position the points within the interval. One approach would be to locate both of them within a small distance on either side of the center of the interval. By this approach, we can effectively cut the interval in half for every two functional evaluations.

The efficiency of a method can be defined as the fraction of the interval eliminated divided by the number of evaluations. By this definition, the efficiency of the approach described in the previous paragraph is

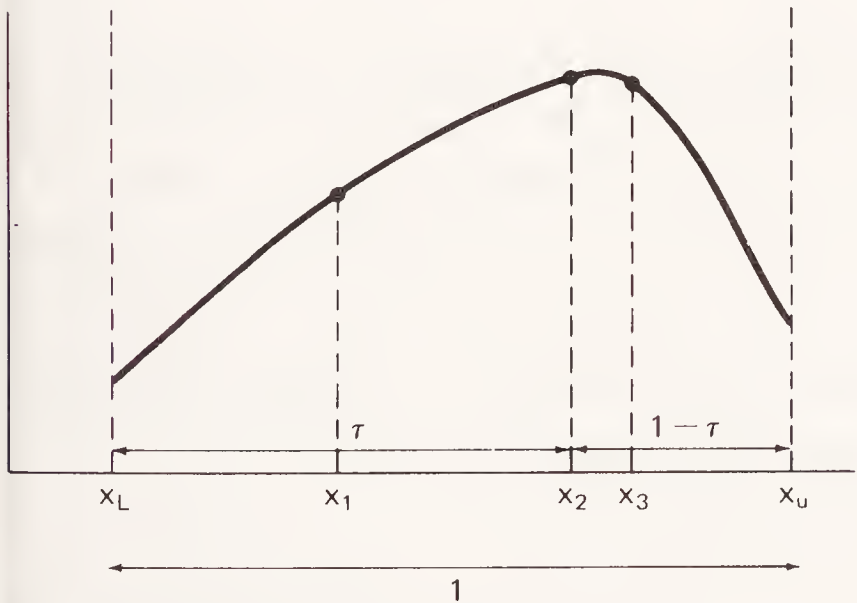
$$\text{eff} = \frac{\text{fraction eliminated}}{\text{functional evaluations}} = \frac{0.5}{2} = 0.25$$

Is there a more efficient approach? Suppose we locate the points so that the interval is trisected, as illustrated in Figure 11-11a. For the curve as drawn, we can eliminate the section (x_L, x_1) and have one point already evaluated within the remaining interval. However, this point is at the center of the remaining interval and not in a position for convenient use.

The most desirable situation would be to locate the points within the interval in such a fashion that the point within the remaining interval is in the correct position for use in the next iteration. Intuitively we know that they must be positioned symmetrically about the center of the interval, as illustrated in Figure



a. Trisection of the interval



b. Locating points for the golden section

Figure 11-11 Positioning of the two intermediate points

372 11-11b. For convenience, suppose we let the length of the original interval be unity. If we let τ equal the ratio of the long subinterval to the total interval, that is,

$$\frac{x_2 - x_L}{x_u - x_L} = \tau$$

then the distance between x_L and x_2 and between x_L and x_u in Figure 11-11b must be τ . The distance between x_L and x_1 and between x_2 and x_u must be $1 - \tau$.

Again referring to Figure 11-11b, interval (x_L, x_1) will be eliminated after the first iteration. If x_2 is to be in proper location for use within the interval (x_1, x_u) , then

$$\frac{x_u - x_2}{x_u - x_1} = \tau$$

From Figure 11-11b, $x_u - x_2 = 1 - \tau$ and $x_u - x_1 = \tau$. Substituting, we have

$$\frac{1 - \tau}{\tau} = \tau$$

or

$$\tau^2 + \tau - 1 = 0$$

Solving for τ , we have

$$\tau = \frac{\sqrt{5} - 1}{2} = 0.618$$

To illustrate the use of these intervals, suppose we apply the golden section to the problem illustrated in Figure 11-8. The length of the original interval is 100. The two intermediate points are to be located at distances $0.618 \times 100 = 61.8$ units from each end, or at $x_1 = 38.2$ and $x_2 = 61.8$, as illustrated in Figure 11-12. Since $f(x_2) > f(x_1)$, we eliminate the interval $(0, 38.2)$. The next point is located at distance $61.8 - 38.2 = 23.6$ from the right end, or at $x_3 = 76.4$. Alternatively, this point could be computed as $0.618 \times (\text{length of interval}) = 0.618 \times 61.8 = 38.2$ units from the left end of the interval, or

$$x_3 = 38.2 + 38.2 = 76.4$$

We now eliminate the interval $(76.4, 100)$ and locate the next point at distance $76.4 - 61.8 = 14.6$ units from the left end, or

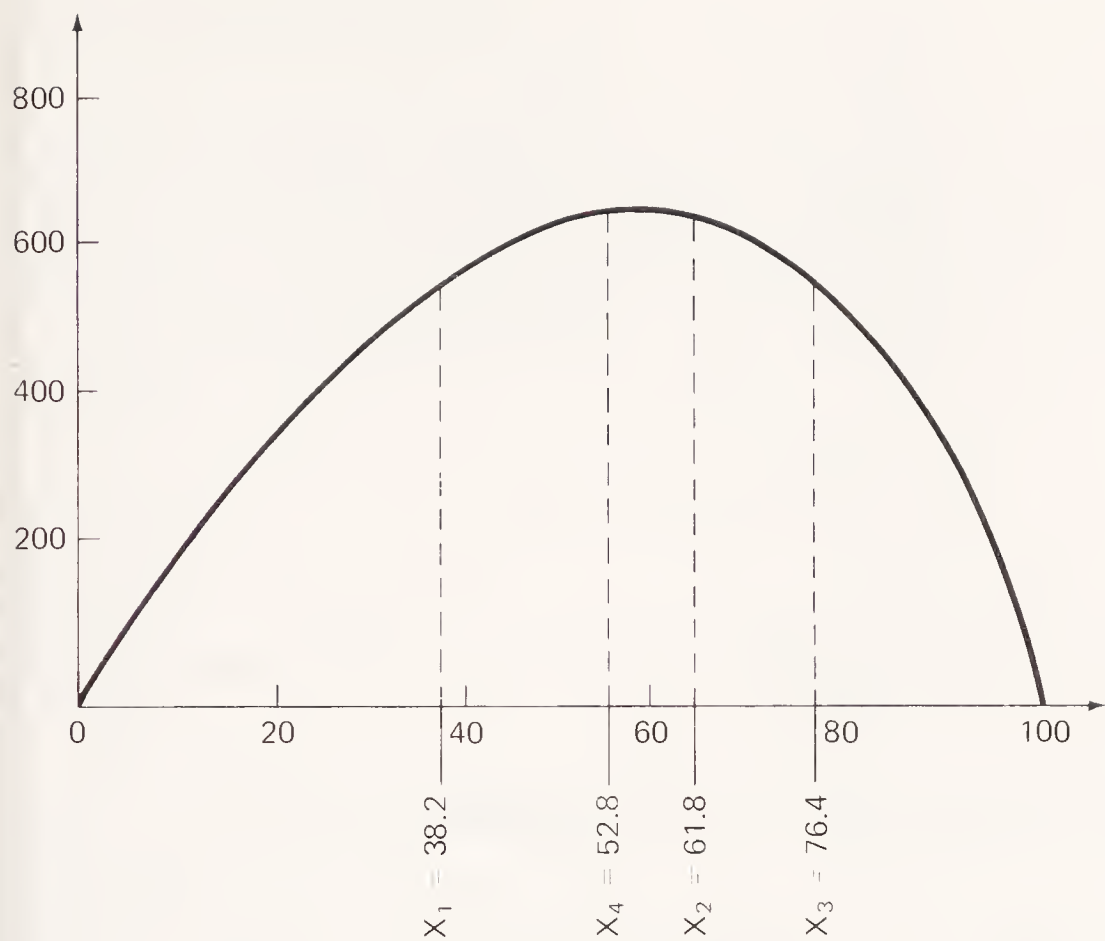


Figure 11-12 Use of the golden section

$$x_4 = 38.2 + 14.6 = 52.8$$

This process can be continued indefinitely.

The flowchart for implementing this algorithm on a digital computer is given in Figure 11-13. The first step is to initialize x_L and x_U (Block 1), then to locate the first two points (Block 2) and then to compute the criterion function values corresponding to these two points (Block 3). In Figure 11-12, it is assumed that a function $F(X)$ is available to compute the criterion function for a given value of X . Block 4 compares these two functional values to determine which interval to eliminate. If $F_1 > F_2$, the interval (x_2, x_u) is to be eliminated. This is done by setting x_u equal to x_2 (Block 5). If $F_2 > F_1$, the interval (x_L, x_1) is to be eliminated, which is accomplished by setting x_L equal to x_1 (Block 8).

For the algorithm in Figure 11-13, the stopping criterion is whenever the length of the interval becomes less than 0.1. Blocks 6 and 9 check for this condition.

If the interval (x_2, x_u) is eliminated, the point x_1 in the old interval should become x_2 for the new interval used on the next iteration. Therefore, Block 7 moves x_1 to x_2 , F_1 to F_2 , and computes a new value for x_1 and F_1 . If the interval (x_L, x_1) is eliminated, the point x_2 in the old interval should become x_1 for the new interval. Therefore, Block 10 moves x_2 to x_1 , F_2 to F_1 , and computes a new value for x_2 and F_2 .

Although it essentially takes one functional evaluation to get the golden section started, the interval is reduced by $1 - 0.618 = 0.382$ for each functional

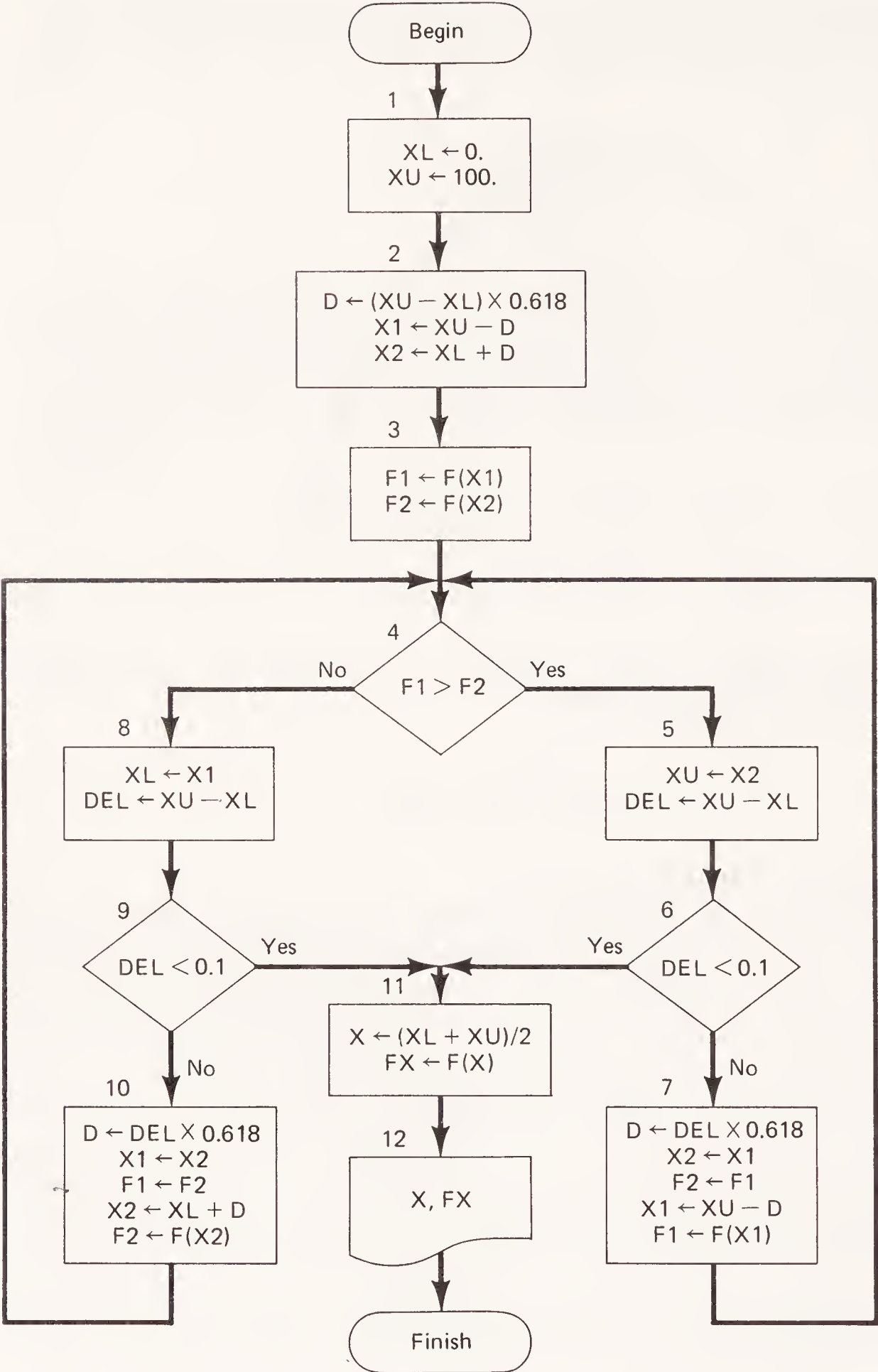


Figure 11-13 Flowchart for golden section

evaluation thereafter. Therefore, the efficiency of the golden section approaches 0.382 as the number of iterations increases.

For one-dimensional optimization problems, the golden section is very close to the optimal procedure. An approach based on the Fibonacci sequence will perform only slightly better, especially if the number of iterations is large.

11.5. INTEGRATION

A student of calculus spends a considerable amount of time studying a process known as *integration*, which is really nothing more than determining the area under a curve. On the digital computer, there are a number of routines for numerically computing areas, some of which we shall examine in this section.

For purposes of our discussion, suppose we want to compute the area under the curve

$$f(x) = 1 + 2x + 3x^2 - x^3$$

between $x = 0$ and $x = 3$, as illustrated in Figure 11-14. A student having completed his calculus course should be able to readily determine that this area is 18.75.

To compute this area numerically, the first step is to divide the interval (0,3) into a number of subintervals. Figure 11-15 illustrates the case where six subintervals are used. The width h of each subinterval is 0.5. The total area under the curve is simply the sum of the areas of the subintervals. We shall therefore consider several different approaches for computing the areas of these subintervals.

The simplest approach to computing the areas is to approximate the areas of the subintervals with rectangles whose heights are the values of the function $f(x)$ evaluated somewhere within the interval. Three cases are illustrated in Figure 11-16:

- 1. Height specified by value of $f(x)$ at the left end (Figure 11-16a). With this approach, the area is approximated by the following equation

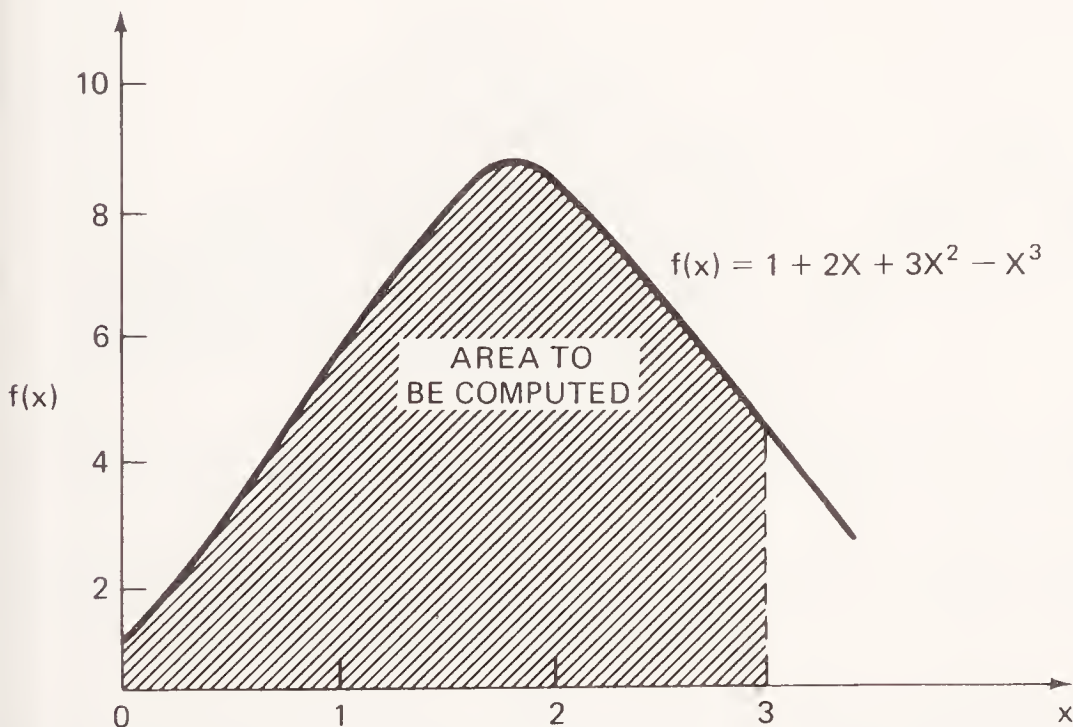


Figure 11-14 Area to be computed

$$\text{Area} = \sum_{i=1}^6 f((i-1)h) \cdot h$$

2. Height specified by value of $f(x)$ at the right end (Figure 11-16b). With this approach, the area is approximated by

$$\text{Area} = \sum_{i=1}^6 f(ih) \cdot h$$

3. Height specified by the value of $f(x)$ at the center of the interval. The area is thus approximated by

$$\text{Area} = \sum_{i=1}^6 f((i-1/2)h) \cdot h$$

From an examination of the graphs in Figure 11-16, the use of the midpoint appears to give the best approximation, which it indeed does.

The flowchart for computing the area by using the values of $f(x)$ at the left end of the interval is given in Figure 11-17. Since six subintervals is generally not a sufficient number, the program reads the number to be used (Block 1) and computes h (Block 2). The loop consisting of Blocks 4 through 7 computes the area of each rectangle and sums them.

To convert the flowchart in Figure 11-17 to use the approximation in Figure 11-16b, only the statement in Block 5 needs to be changed to

$$X = J \times H$$

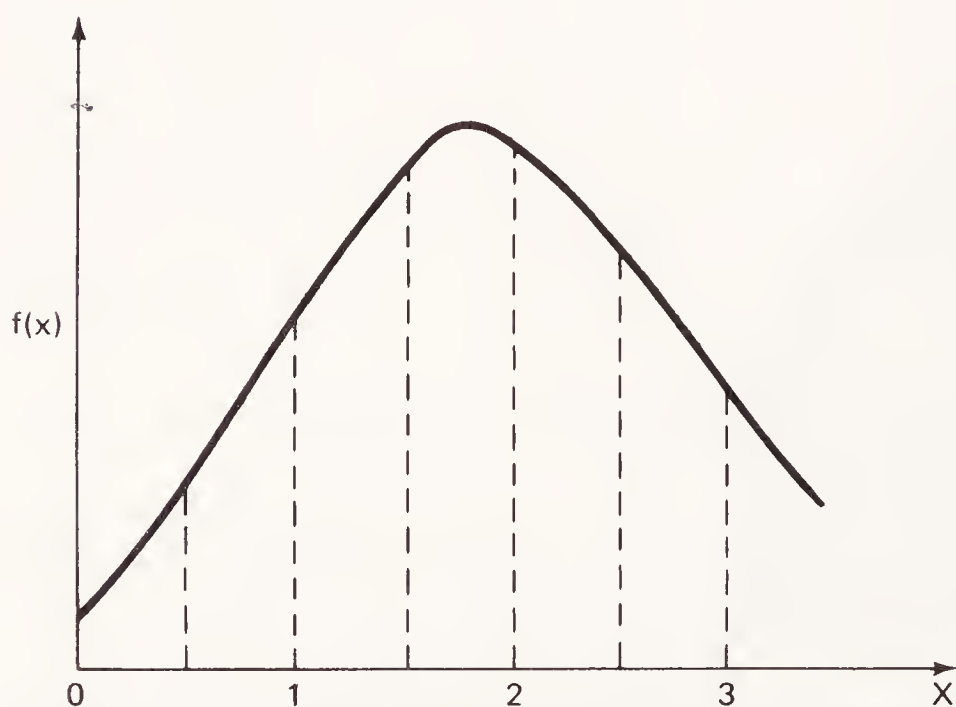
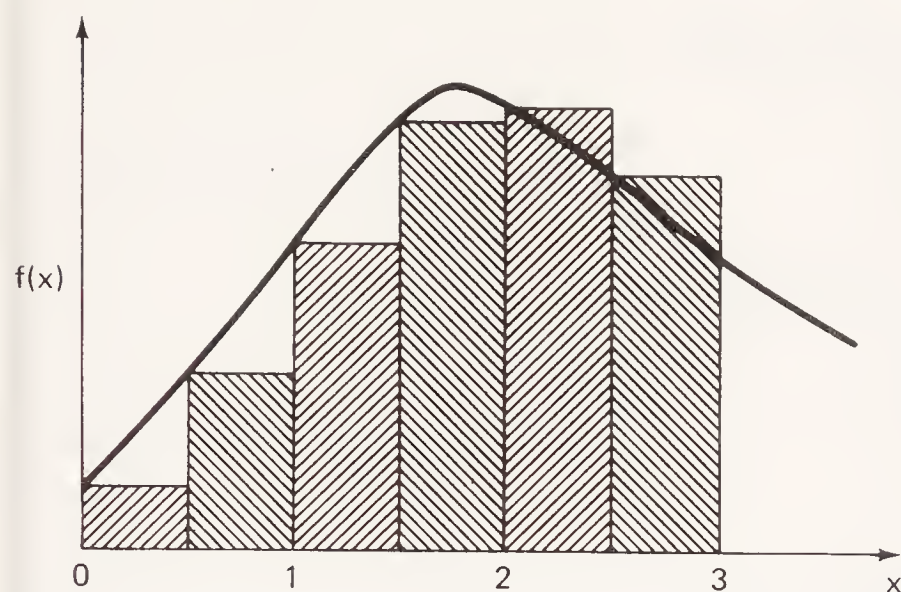
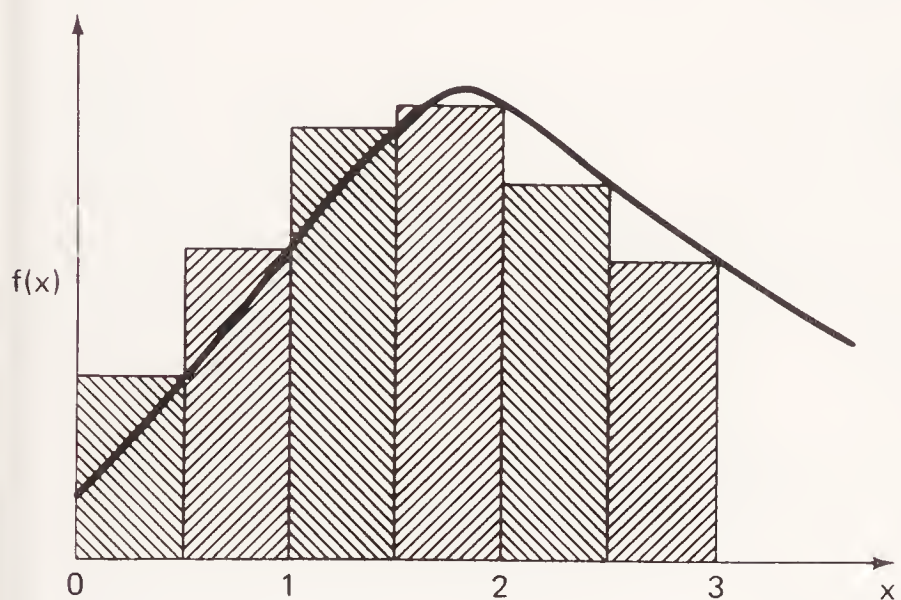


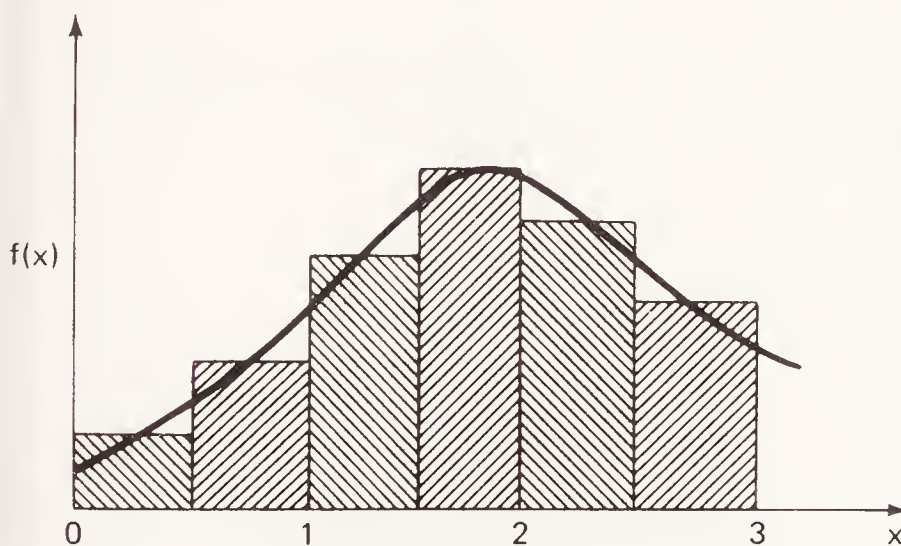
Figure 11-15 Division of the interval (0,3) into six subintervals



a. Use of value of $f(x)$ at left-hand end to determine height of rectangle



b. Use of value of $f(x)$ at right-hand end to determine height of rectangle



c. Use of midpoint to determine height of the rectangle

Figure 11-16 Approximating area under the curve by rectangles

To use the approximations in Figure 11-16c, this statement should be

$$X = (J - 0.5) \times H$$

In a subsequent paragraph, we shall return to the question of the quality of these approximations.

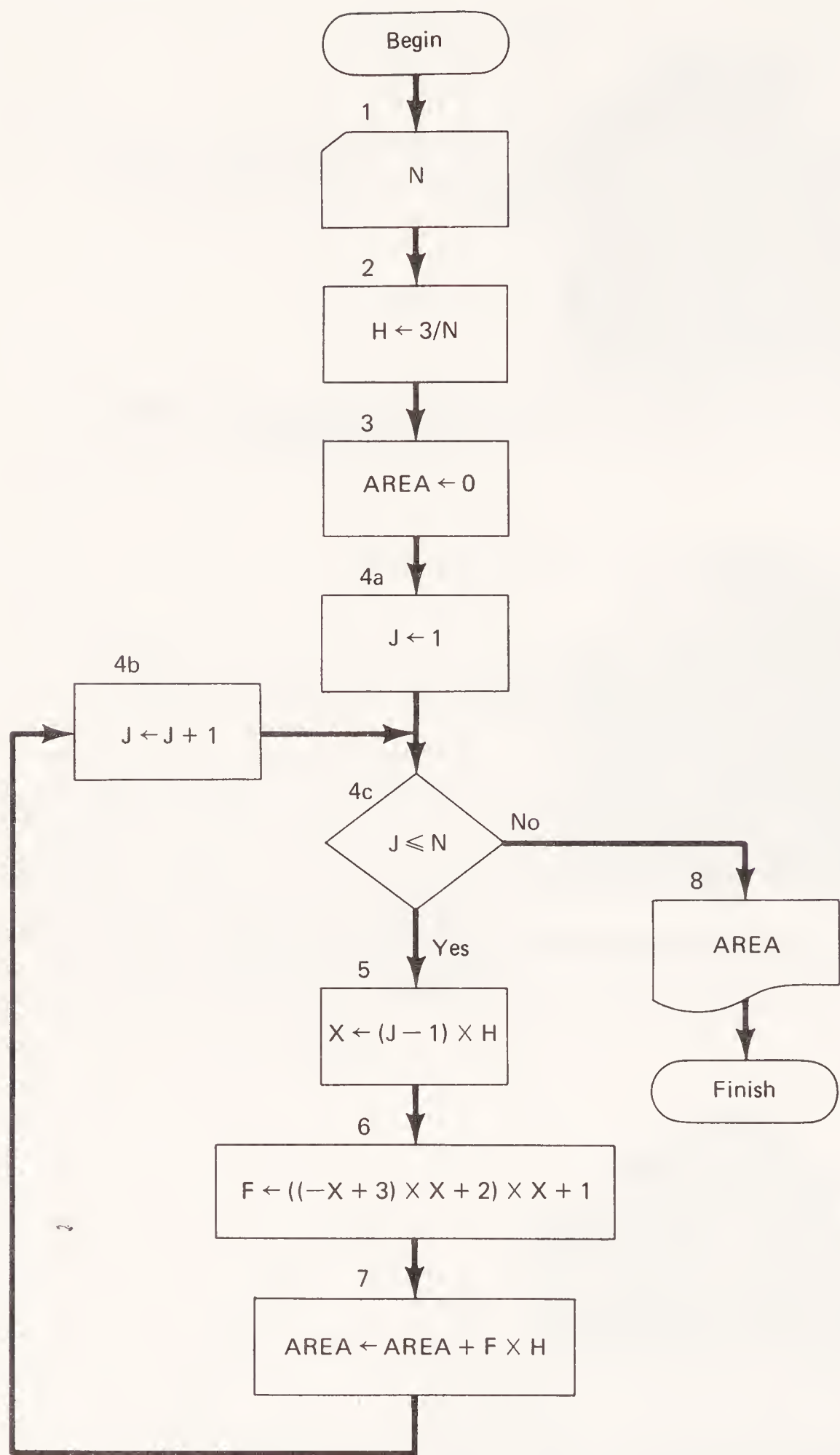


Figure 11-17 Computation of area under curve using approximation of Figure 10-16a

Another method for approximating the area of the curve is illustrated in Figure 11-18. This method, known as the *trapezoid rule*, determines the area under a straight line drawn between the values of $f(x)$ at each end of the increment. The area for the first subinterval is

$$\frac{f(0) + f(h)}{2} \cdot h$$

For the second, it is

$$\frac{f(h) + f(2h)}{2} \cdot h$$

The total area is given by

$$\text{Area} = \sum_{i=1}^6 \frac{f((i-1) \cdot h) + f(ih)}{2} \cdot h$$

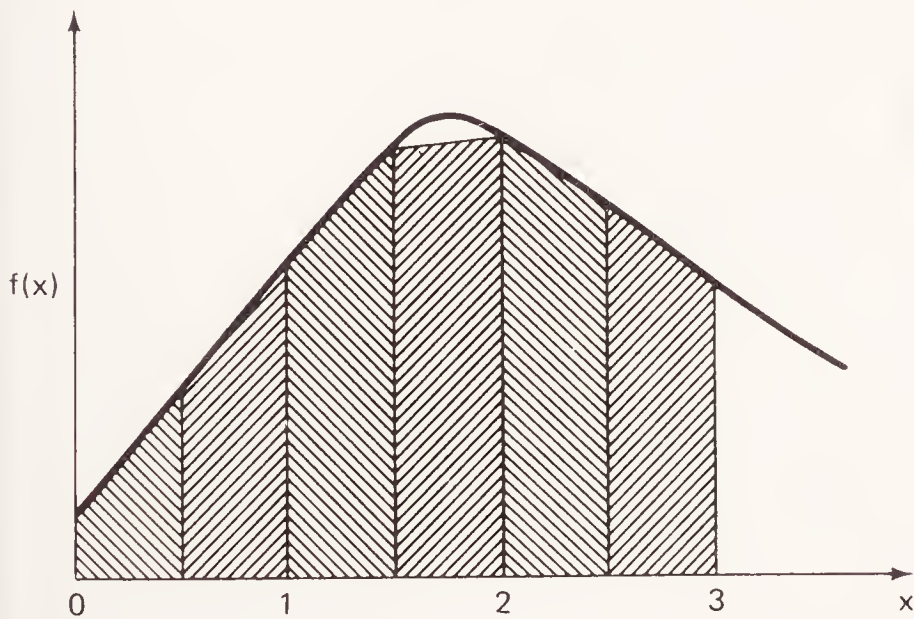
The flowchart of the algorithm for performing this computation is given in Figure 11-18b.

An even more accurate integration technique is known as *Simpson's rule*. Whereas the trapezoidal rule approximated the area by drawing a straight line between two adjacent points, Simpson's rule approximates the area by fitting the constants in the equation

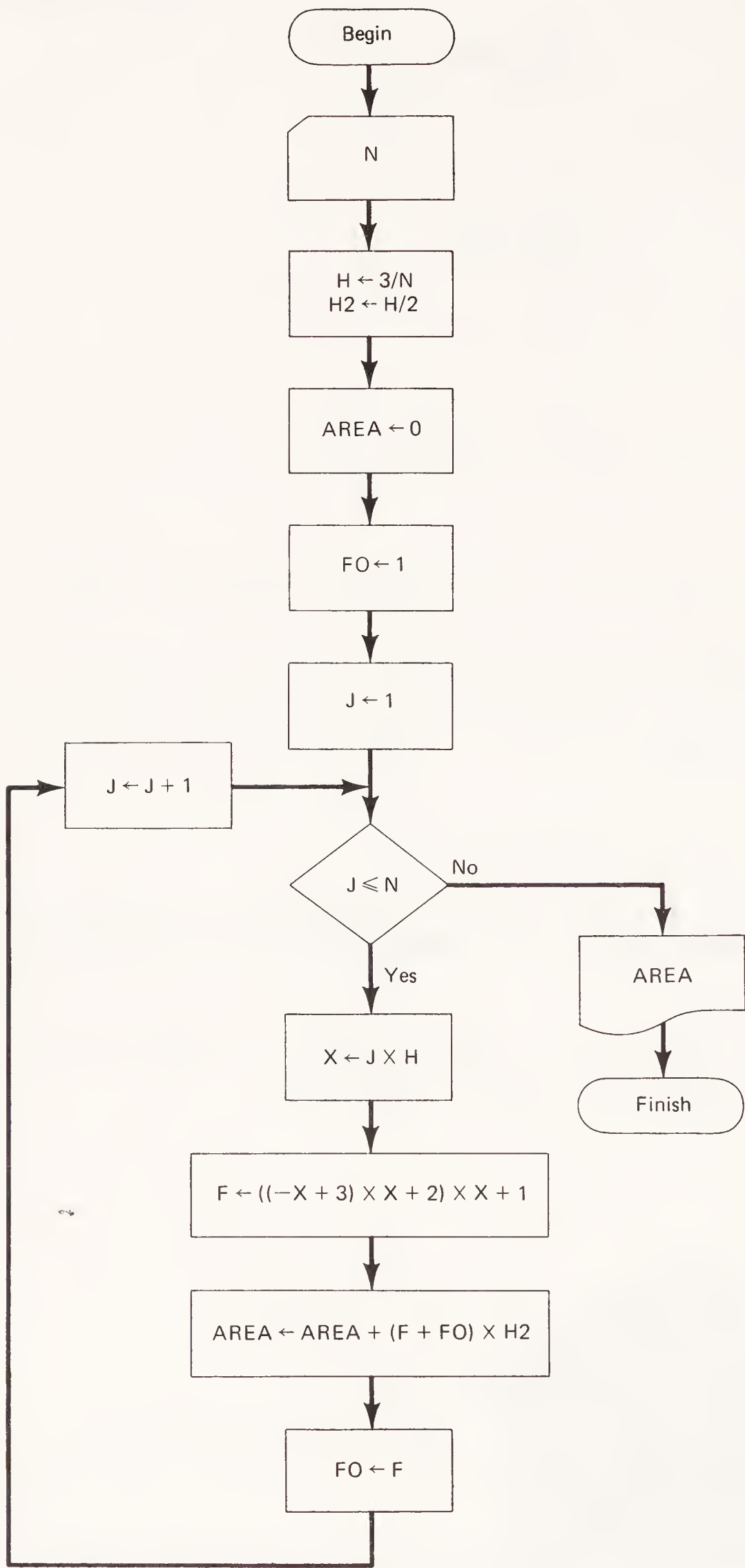
$$y = a + b (x + c)^2$$

to three adjacent points. In effect, a curved line (specifically, a parabola), is fit and the area computed analytically. The resulting equation for integration is as follows:

$$\begin{aligned} \text{Area} = 1/3 [&f(0) + 4f(h) + 2f(2h) + 4f(3h) + 2f(4h) + \cdots + 4f((n-3)h) \\ &+ 2f((n-2)h) + 4f((n-1)h) + f(nh)] \end{aligned}$$



a. Geometric representation of the trapezoidal approximation
Figure 11-18 Integration via the trapezoid rule



b. Flowchart

Figure 11-18 (Continued)

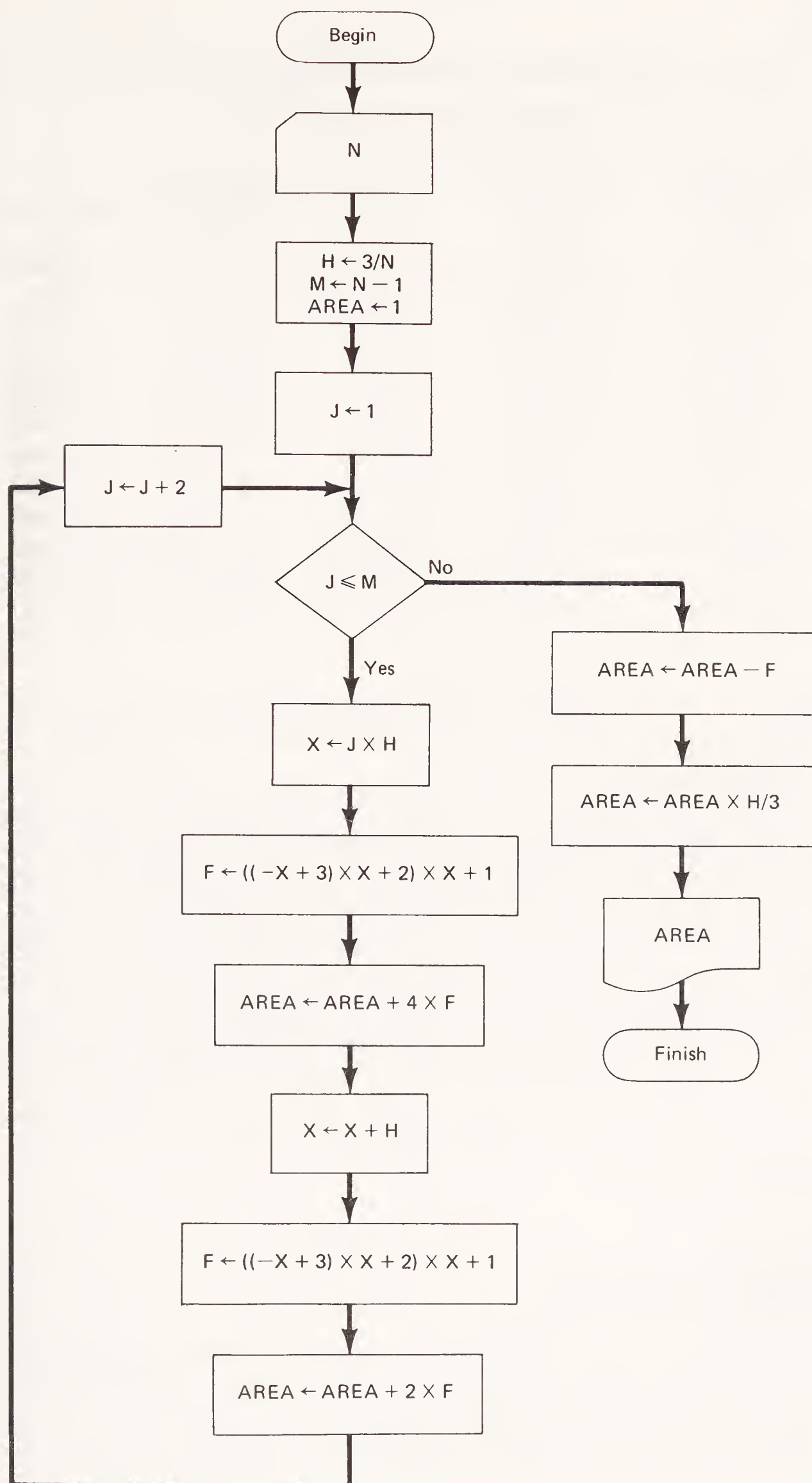


Figure 11-19 Flowchart of algorithm to integrate via Simpson's rule

Table 11-2 Accuracy of Numerical Integration Techniques

Number of intervals	INTEGRATION TECHNIQUE				
	Rectangular (Figure 10-16a)	Rectangular (Figure 10-16b)	Midpoint (Figure 10-16c)	Trapezoid (Figure 10-18)	Simpson's Rule
5	16.6800	20.2800	18.8850	18.4800	
10	17.7825	19.5825	18.7838	18.6825	18.7500
20	18.2831	19.1831	18.7584	18.7331	18.7500
50	18.5673	18.9273	18.7514	18.7473	18.7500
100	18.6593	18.8393	18.7503	18.7493	18.7500
200	18.7048	18.7948	18.7501	18.7498	18.7500
500	18.7320	18.7680	18.7500	18.7500	18.7500
1000	18.7410	18.7590	18.7500	18.7500	18.7500
2000	18.7455	18.7545	18.7500	18.7500	18.7500
5000	18.7482	18.7518	18.7500	18.7500	18.7500
10000	18.7491	18.7509	18.7500	18.7500	18.7500

In this equation, *n* must be an even integer; that is, there must be an even number of increments. As seen from the flowchart in Figure 11-19, this method is also quite straightforward.

Table 11-2 gives an indication of the quality of the approximations resulting from each of the five approaches indicated in this section. In general, the two approximations to the area using rectangles whose height is the value of the function at either end (Figures 11-16a and b) are about equivalent with regard to error, but poorest of all. Using either a rectangle whose height is the value of the function at the midpoint or using the trapezoid rule is significantly better, and using Simpson's rule is still even better. As Table 11-2 indicates, however, any of the approximations can be used provided the width of the subinterval is made small enough. In general, use of the more accurate techniques will give the same degree of accuracy with a larger subinterval than will be required when using the less accurate methods. Use of a larger subinterval means less work for the computer and thus is preferred.

11.6. GAUSS REDUCTION

The solution of sets of simultaneous linear algebraic equations is frequently encountered in business, science, engineering, etc. To solve such sets of equations, the approach commonly taught is to solve one equation for one of the variables and substitute into the other equation(s). For example, to solve the set of equations

2x + y = 4

x + 2y = 5

we could first solve the second equation for x , obtaining

$$x = 5 - 2y$$

Substituting into the first equation, we obtain

$$\begin{aligned} 2(5 - 2y) + y &= 4 \\ -3y &= -6 \\ y &= 2 \end{aligned}$$

Now computing x , we have

$$x = 5 - 2y = 1$$

This approach can be applied to any set of simultaneous linear algebraic equations. For example, to solve the set

$$\begin{aligned} 2x + y - z &= 5 \\ x + 2y + 2z &= 3 \\ -2x - y + 4z &= -8 \end{aligned}$$

we would solve one of the equations for one of the variables and substitute into the other two, thereby obtaining a set containing two equations and two unknowns. Solving one of these for one variable and substituting into the other gives one equation and one unknown, which can be solved. Thus, for every substitution, the number of equations and number of variables are each reduced by one.

The Gauss reduction is an algorithm for effectively accomplishing the same process with the computer. To illustrate how this algorithm functions, suppose we apply it to the 2×2 set of equations solved previously:

$$\begin{aligned} 2x + y &= 4 \\ x + 2y &= 5 \end{aligned}$$

The first step is to multiply the top equation by a constant and add to the second equation. The constant is chosen such that the coefficient of x in the final result is zero. For this set of equations, the proper constant is $-1/2$. Multiplying the top equation by $-1/2$ and adding to the second equation gives the following set:

$$\begin{aligned} 2x + y &= 4 \\ 0x + 1.5y &= 3 \end{aligned}$$

The second equation now contains only one unknown, which can be solved to give

$$y = 2$$

This value can be back-substituted into the top equation to obtain

$$2x + 2 = 4$$

$$x = 1$$

Thus, the procedure consists of a reduction followed by back-substitution.

As a further example, suppose we apply the Gauss reduction to the following 3×3 set:

$$\begin{array}{rrcr} 2x + & y - & z = & 5 \\ & x + 2y + 2z = & 3 \\ -2x + & y + 4z = & -4 \end{array}$$

If we multiply the top equation by $-1/2$ and add to the second, we obtain

$$\begin{array}{rrcr} 2x + & y - & z = & 5 \\ 0 & + 1.5y + 2.5z = & 0.5 \\ -2x + & y + 4z = & -4 \end{array}$$

Next, we multiply the top equation by $+1$ and add to the third to obtain

$$\begin{array}{rrcr} 2x + & y - & z = & 5 \\ 0 & + 1.5y + 2.5z = & 0.5 \\ 0 & + 2y + 3z = & 1 \end{array}$$

The last two equations contain only two unknowns. Applying the Gauss reduction procedure to them, we multiply the second equation by $-2/1.5$ and add to the third to obtain:

$$\begin{array}{rrcr} 2x + & y - & z = & 5 \\ 0 & + 1.5y + 2.5z = & 0.5 \\ 0 & + 0 - 1/3z = & 1/3 \end{array}$$

The third equation can be solved to give

$$z = -1$$

This value can be substituted into the second equation to obtain

$$\begin{array}{rr} 1.5y - 2.5 = & 0.5 \\ y & = 2 \end{array}$$

The values for y and z can now be substituted into the top equation to give

$$\begin{aligned} 2x + 2 + 1 &= 5 \\ x &= 1 \end{aligned}$$

This completes the solution.

To express the set of equations in a convenient form for computer solution, a 2×2 set of equations would be represented as follows:

$$\begin{aligned} a_{1,1} x_1 + a_{1,2} x_2 &= b_1 \\ a_{2,1} x_1 + a_{2,2} x_2 &= b_2 \end{aligned}$$

With this representation,

- x_j = j th unknown
- b_i = constant term for the i th equation
- $a_{i,j}$ = coefficient of the j th unknown in the i th equation

A 3×3 set would be represented as follows:

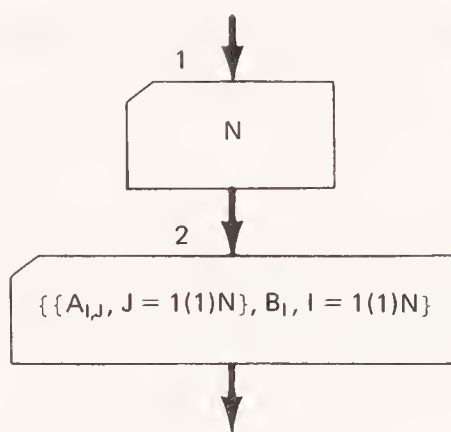
$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 &= b_3 \end{aligned}$$

This representation can be applied to equations of any order.

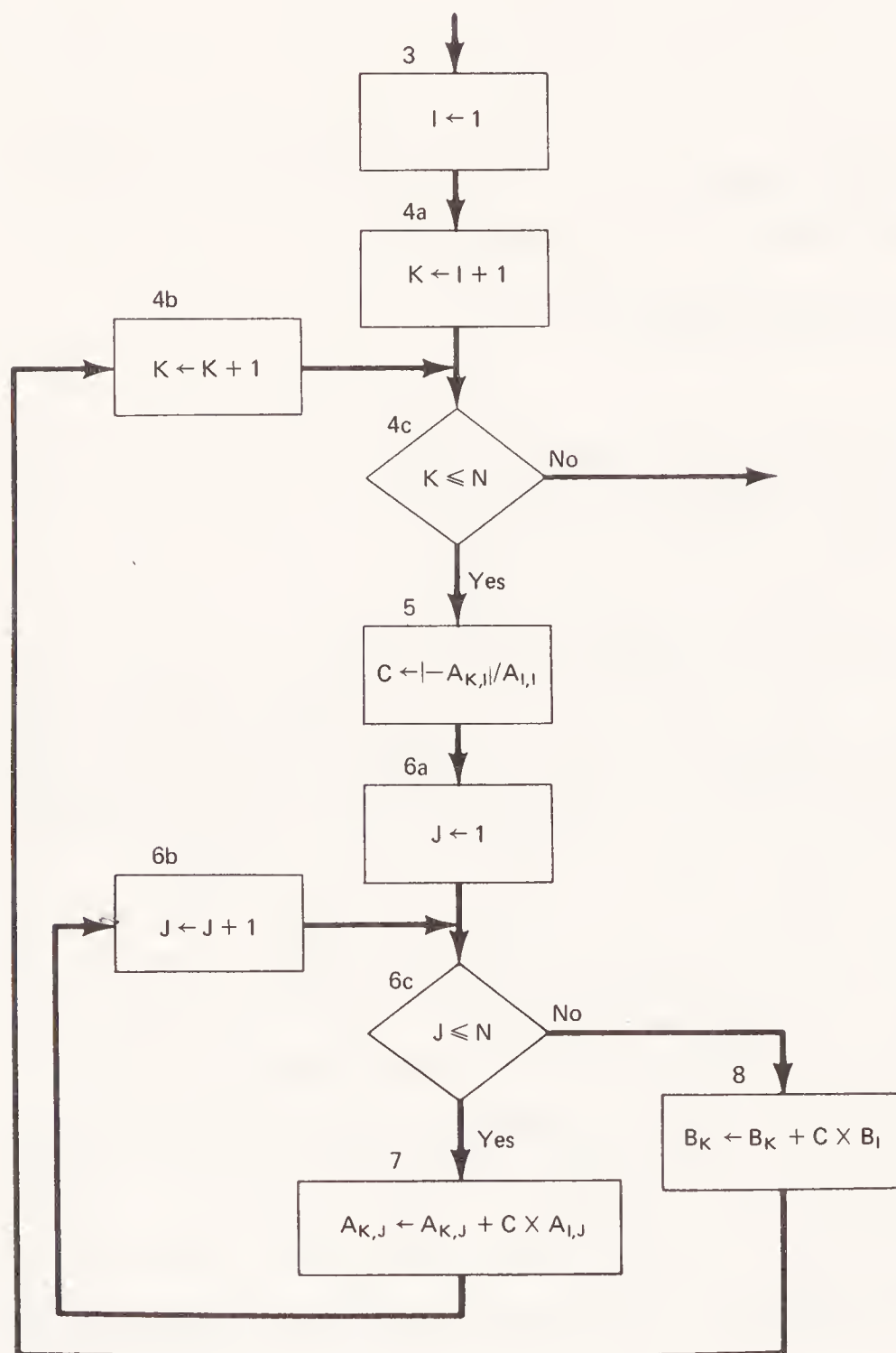
For the general case, we might input the data as follows. The first card contains the value of N , the number of equations in the set. The following card contains the values for all coefficients and the constant term in the first equation. The next card contains the corresponding values for the second equation, etc. The input statements would be as given in Figure 11-20a.

To accomplish the first reduction step (that is, to obtain zeros for the coefficients of x_1 in all but the first equation), the logic in Figure 11-20b can be used. Block 3 sets I equal to 1, which indicates that we shall be multiplying the first equation by a constant and adding to each of the remaining equations. Block 4 is an iteration counter that causes the reduction operation to be applied to each of the remaining equations. The index K starts at $I + 1$ and continues through N . Block 5 computes the coefficient by which equation 1 will be multiplied. The loop (Blocks 6 and 7) computes new values for the coefficients in the reduced equations, and Block 8 computes a new value for the constant term.

If Block 3 is replaced by an iteration counter to vary I from 1 to $N - 1$ and the flowcharts in Figure 11-20 combined, a complete algorithm for the reduction step is obtained, as given in Figure 11-21.



a. Input statements



b. Flowchart

Figure 11-20 Elements of the Gauss reduction algorithm

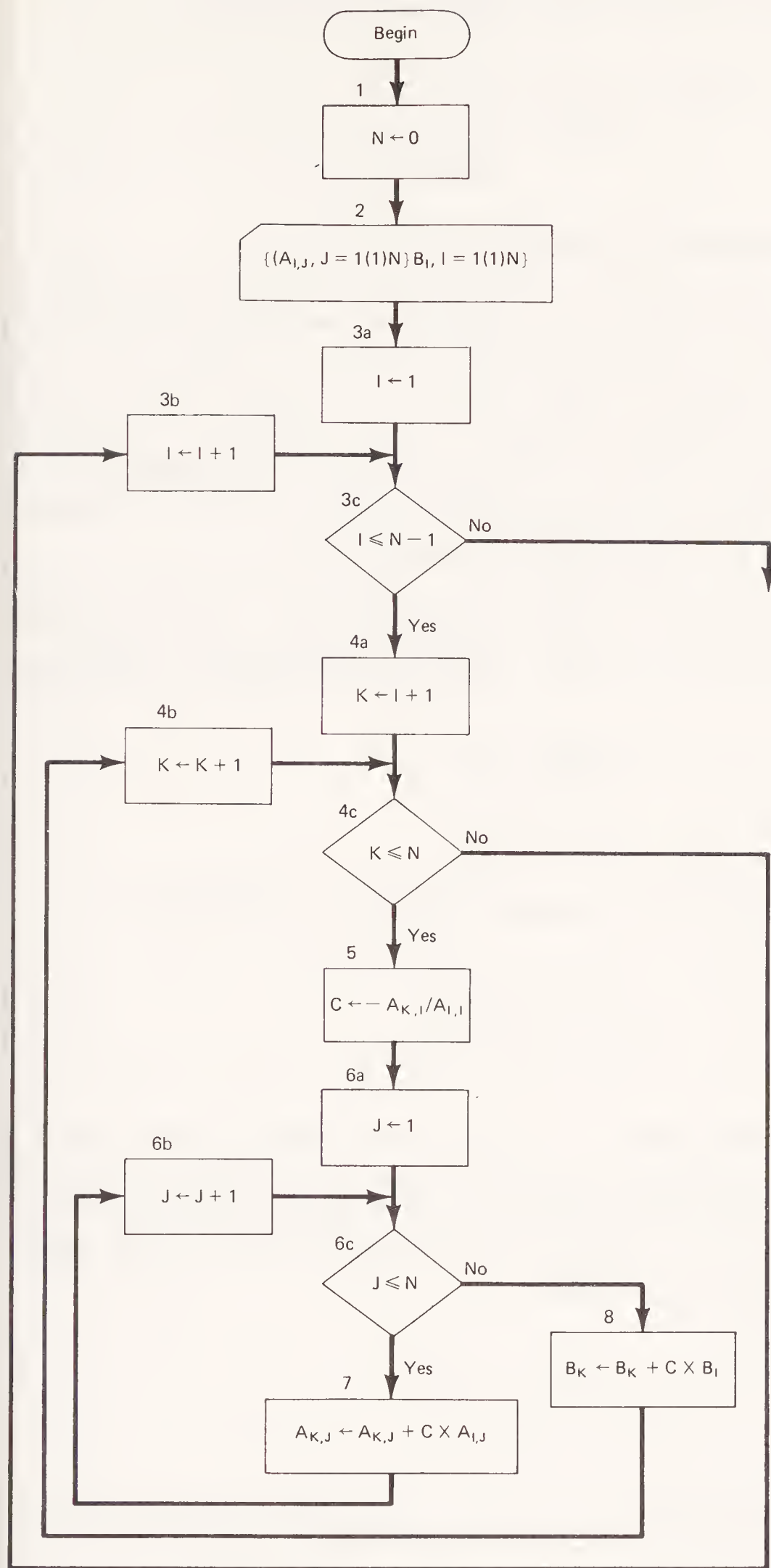


Figure 11-21 Complete algorithm for the reduction step

388 For a 3×3 set of equations, the result would appear as follows:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1 \\ a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \\ a_{3,3}x_3 &= b_3 \end{aligned}$$

We are now ready to begin the back-substitution process.

The first step is to solve the last equation for the n th unknown, which is accomplished by Block 9 in Figure 11-22. We then start at the $(n - 1)$ th row and back-substitute. Using K as the index for the row, we set K equal to $N - 1$ in Block 10. Block 11 initializes variable SUM to the constant term for that equation, and the loop consisting of Blocks 12 and 13 subtracts the product of each previously computed x and its coefficient. Block 14 divides the result by the diagonal coefficient to determine the value of x_k . Blocks 15 and 16 decrease K by 1 and check to see if we have processed all the equations.

While a Gauss reduction consisting of the algorithms in Figure 11-21 and 11-22 would function properly for many sets of equations, conditions exist for which it will not perform. For example, consider the following set of equations:

$$\begin{aligned} x + 2y - z &= 4 \\ x + 2y + z &= 2 \\ x + y + z &= 1 \end{aligned}$$

Applying the first step of the Gauss reduction produces the following set:

$$\begin{aligned} x + 2y - z &= 4 \\ 0 + 0 + 2z &= -2 \\ 0 + -y + 2z &= -3 \end{aligned}$$

If we attempt to directly apply the next step of the Gauss reduction, we find that the coefficient of 0 for the y term in the second equation makes it impossible to reduce the third equation. In the flowchart in Figure 11-21, a division by zero would occur in Block 5. If we simply interchange the second and third equations in the set, we obtain

$$\begin{aligned} x + 2y - z &= 4 \\ 0 - y + 2z &= -3 \\ 0 + 0 + 2z &= -2 \end{aligned}$$

and the reduction procedure can be continued, although for this particular case, we are finished.

In general, the preferred procedure is to rearrange the equations such that the one with the largest (in magnitude) coefficient is used to reduce the remaining equations. To accomplish this, the algorithm in Figure 11-23 should be inserted

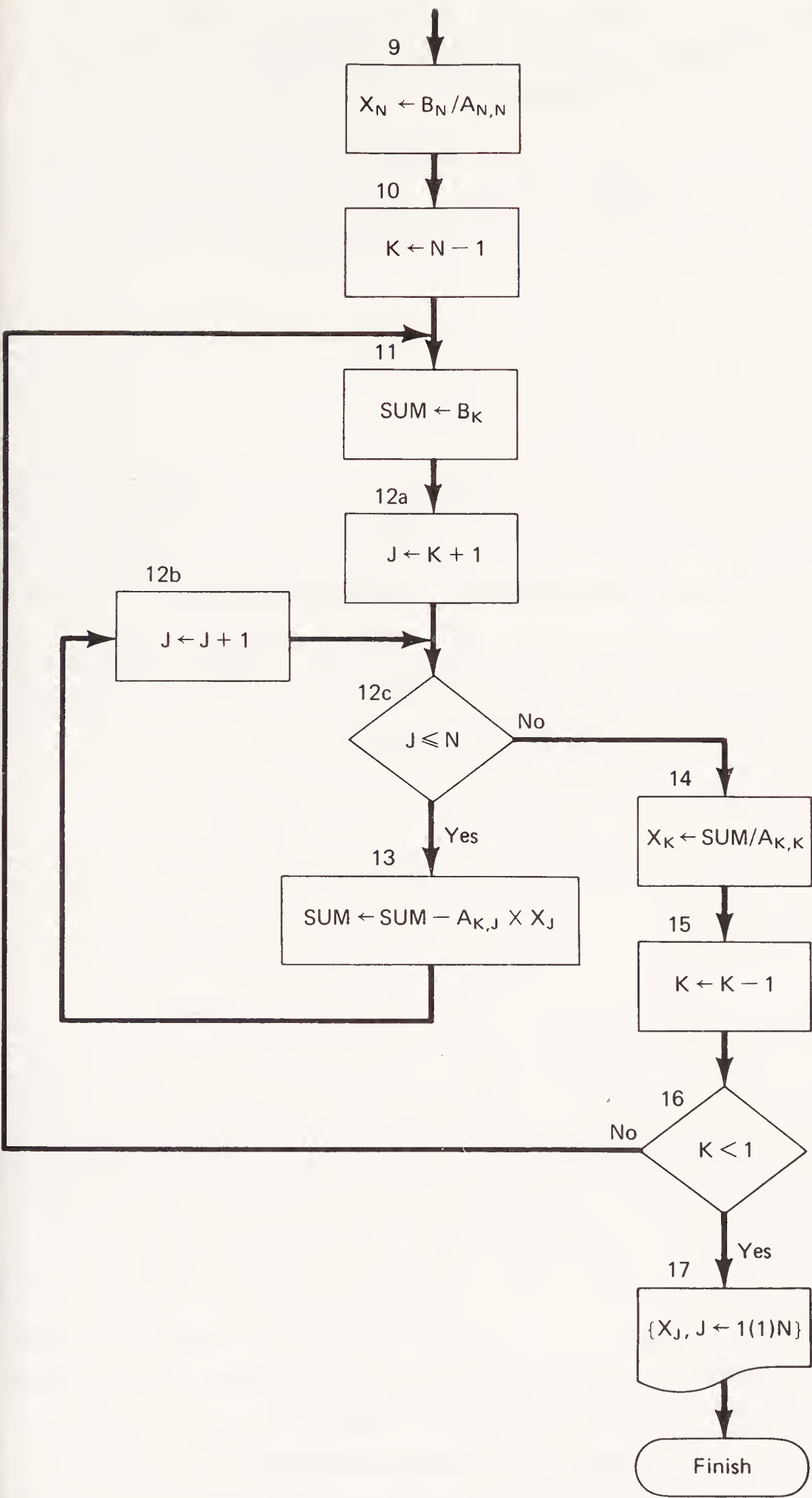


Figure 11-22 Back-substitution algorithm

390 between Blocks 3 and 4 in Figure 11-21. In this algorithm, Blocks 19, 20, 21, and 22 determine which row has the largest coefficient for this particular reduction step. Block 23 interchanges the constant terms; Blocks 24 and 15 interchange the coefficients.

With the algorithm in Figure 11-23 inserted into the algorithm in Figure 11-21, the value of C computed in Block 5 should always be less than one in magnitude. It is possible, however, for the following type of situation to be encountered:

$$\begin{aligned}x + y + z &= 1 \\0 + 0 + z &= 2 \\0 + 0 + z &= 3\end{aligned}$$

This set of equations has no solution. Using the algorithm in Figure 11-23, this condition can be detected by checking the magnitude of the largest coefficient located. If this largest coefficient is too small, the reduction step should be discontinued and a message printed saying that the set does not have a unique solution.

Although the row interchange as described above is a definite improvement over the basic reduction procedure, it is not entirely satisfactory. For example, the following two sets of equations have the same solution:

Set A	Set B
$9X_1 + 6X_2 - 2X_3 = 5$	$0.9X_1 + 0.6X_2 - 0.2X_3 = 0.5$
$7X_1 + 4X_2 - 5X_3 = -2$	$7X_1 + 4X_2 - 5X_3 = -2$
$-X_1 + 4X_2 + 8X_3 = 9$	$-X_1 + 4X_2 + 8X_3 = 9$

The only difference between the two sets is that the first equation in Set A has been divided by 10 to obtain Set B.

Applying the approach described above, the first reduction step would be applied to the equations in Set A as they appear. But for Set B, the first two equations would be interchanged before the reduction step is begun. In view of the fact that the two sets of equations are equivalent, one would suspect that the same operations should be applied to both, which does not occur. This is because we have failed to *equilibrate* the equations, a procedure we shall describe below.

The situation that occurred for the two sets of equations presented above will not occur if we simply divide each equation by the coefficient with the largest absolute value. In doing this, both sets of equations become

$$\begin{aligned}X_1 + 0.667X_2 - 0.222X_3 &= 0.556 \\X_1 + 0.571X_2 - 0.714X_3 &= 0.286 \\-0.125X_1 + 0.500X_2 + X_3 &= 1.125\end{aligned}$$

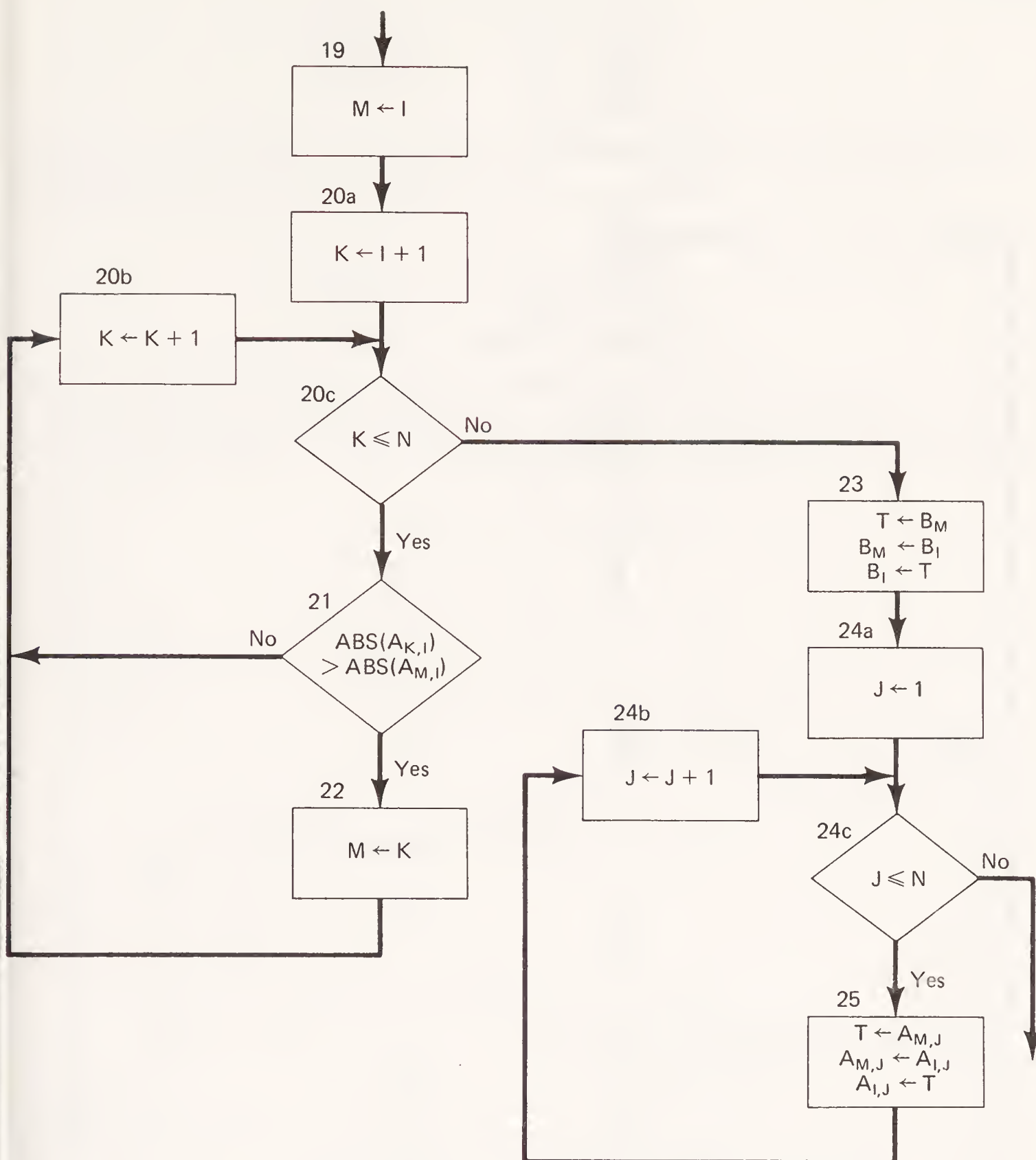


Figure 11-23 Flowchart of algorithm for row interchange

Performing the first step in the reduction produces the following set of equations:

$$X_1 + 0.667X_2 - 0.222X_3 = 0.556$$

$$0X_1 - 0.096X_2 - 0.492X_3 = 0.842$$

$$0X_1 + 0.583X_2 + 0.972X_3 = 1.195$$

The last two equations are again divided by the coefficient with the largest absolute value, giving

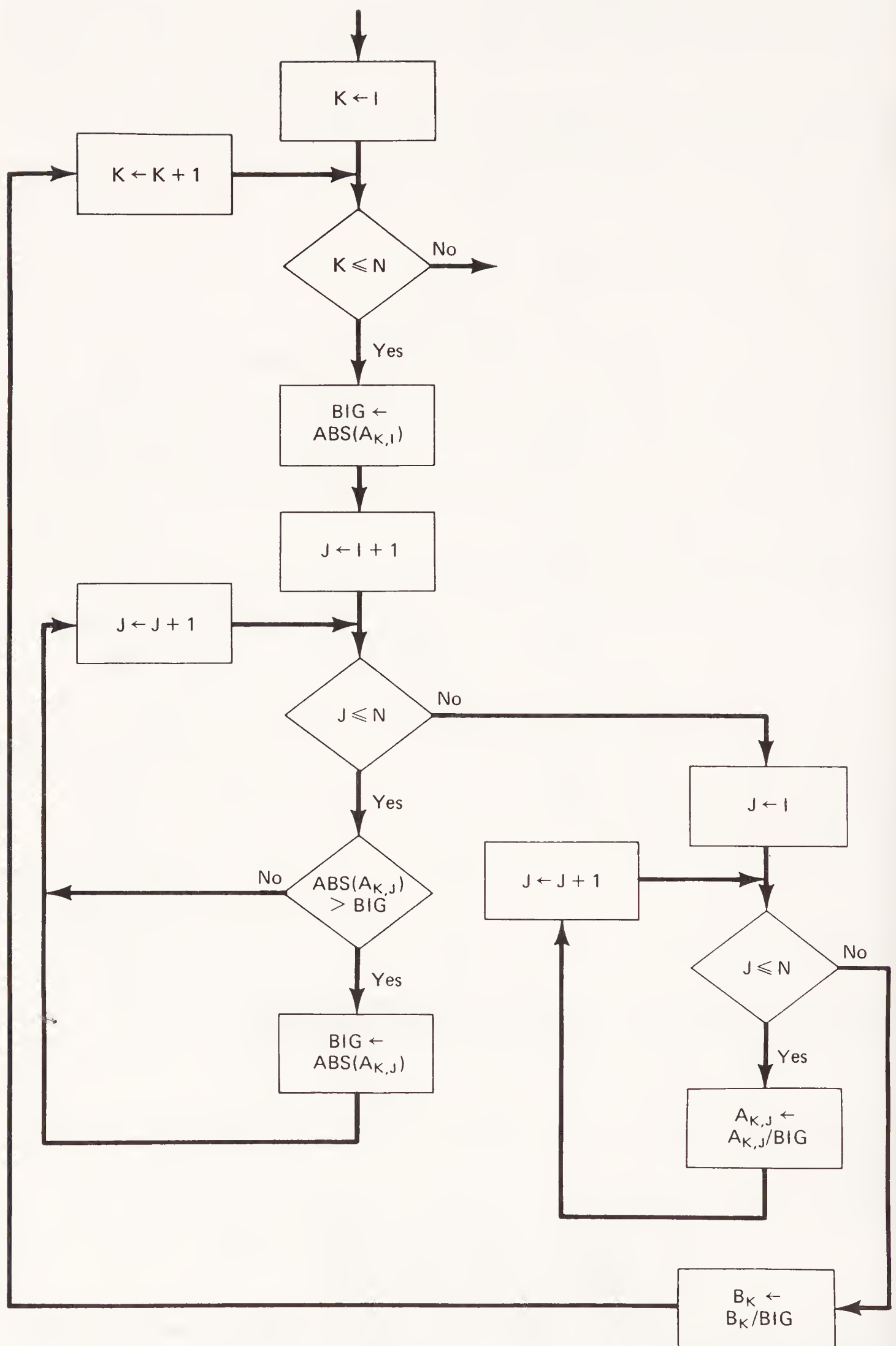


Figure 11-24 Flowchart of algorithm to equilibrate rows

$$\begin{aligned} X_1 + 0.667X_2 - 0.222X_3 &= 0.556 \\ 0.195X_2 + X_3 &= 1.711 \\ 0.599X_2 + X_3 &= 1.229 \end{aligned}$$

The last two equations should be interchanged so that the largest coefficient on X_2 appears in the second equation, that is,

$$\begin{aligned} X_1 + 0.667X_2 - 0.222X_3 &= 0.556 \\ 0.599X_2 + X_3 &= 1.229 \\ 0.195X_2 + X_3 &= 1.711 \end{aligned}$$

Applying the reduction step gives the reduced set of equations.
The flowchart of the algorithm to locate the largest coefficient in each equation and divide by that coefficient is given in Figure 11-24. This should

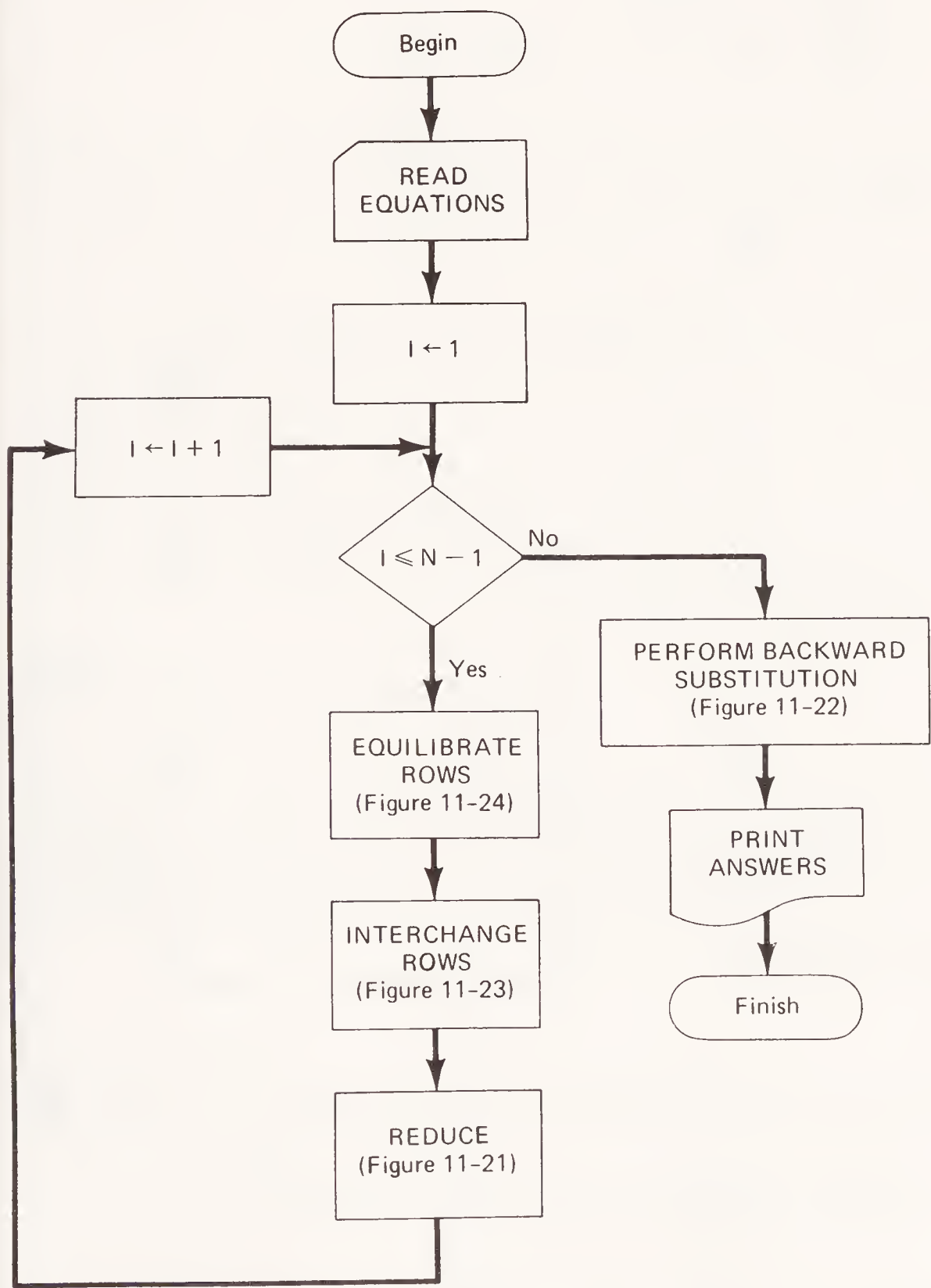


Figure 11-25 Gauss elimination with equilibration

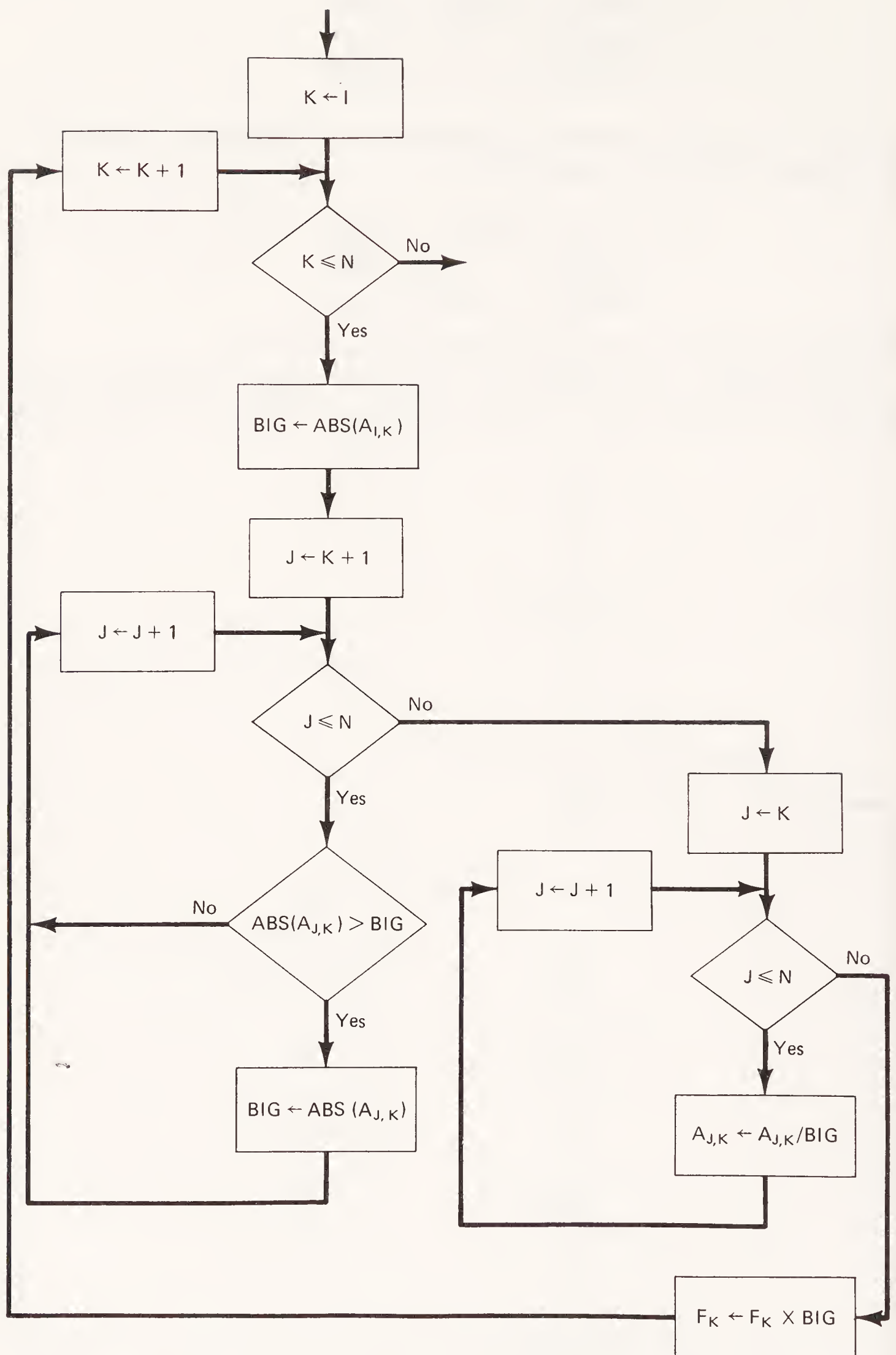


Figure 11-26 Flowchart of algorithm to equilibrate by columns

precede the interchanging of the rows, as illustrated by the overall flowchart in Figure 11-25.

The procedure can be further improved by equilibrating the columns in addition to the rows. When rows in the first step for the above set of equations were equilibrated, the result was

$$\begin{aligned} X_1 + 0.667X_2 - 0.222X_3 &= 0.556 \\ X_1 + 0.571X_2 - 0.714X_3 &= 0.286 \\ -0.125X_1 + 0.500X_2 + X_3 &= 1.125 \end{aligned}$$

Note that both the first and third column contain at least one unity coefficient. In the second column, however, the largest (in magnitude) coefficient is 0.667. If this column is divided by 0.667, the result is

$$\begin{aligned} X_1 + X_2 - 0.222X_3 &= 0.556 \\ X_1 + 0.856X_2 - 0.714X_3 &= -0.286 \\ -0.125X_1 + 0.750X_2 + X_3 &= 1.125 \end{aligned}$$

and contains a unity coefficient in column 2. However, this operation affects the resulting value of X_2 . To determine the solution to the original set of equations, the resulting value of X_2 must be multiplied by 0.667.

The flowchart of the algorithm to equilibrate by columns is given in Figure 11-26, and should be inserted into the overall process immediately following the “Equilibrate by rows” in Figure 11-25. The flowchart in Figure 11-26 utilizes a single-subscripted array F to record the changes made in the coefficients in each column. Each element of this array should be initially set equal to unity, and each time a column of coefficients is divided by a number, the corresponding element in F should be multiplied by that number. Just prior to printing, each element of X should be multiplied by the corresponding element of F .

11.7. LINEAR REGRESSION

In many endeavors of an experimental nature, the investigator collects data pertinent to some event or phenomenon. For example, one might try to ascertain in what manner the price of a book depends upon the number of pages in the book. A typical set of data might appear as follows:

Number of pages	Price
462	\$14.95
127	8.40
247	10.20
512	16.20
324	13.47
410	15.50

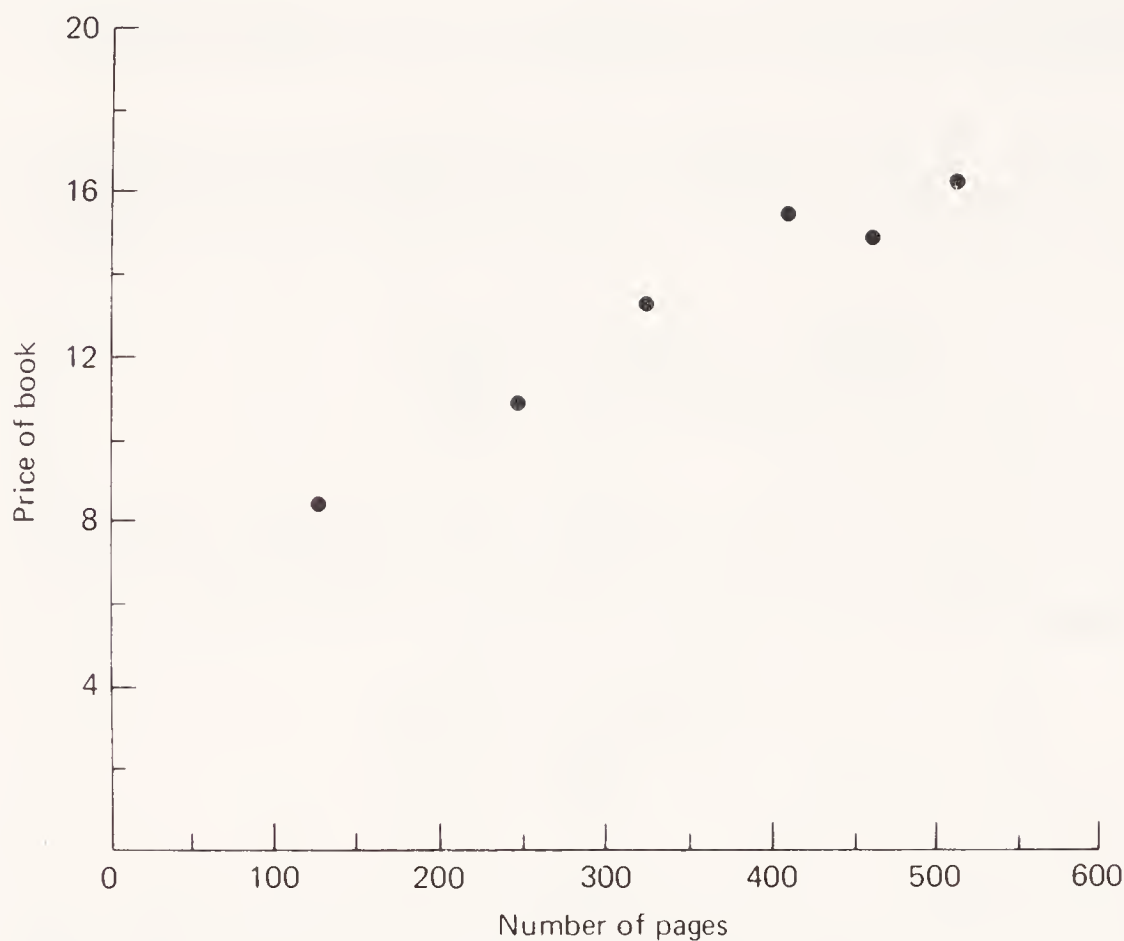


Figure 11-27 Book price *versus* number of pages

Each entry in this set of data is called an *observation*; that is, the fact that a given book contains 462 pages and sells for \$14.95 constitutes an observation.

Either during or after collecting the data, the investigator may at his discretion plot the observations to obtain a result such as in Figure 11-27 for the above data. In effect, these data provide information concerning the relationship of one variable (called the *dependent* variable) upon another variable (called the *independent*, variable). In our example, the independent variable is the pages in the book and the dependent variable is the price of the book.

The ultimate objective of many of these undertakings is to obtain an equation relating the dependent variable to the independent variable. In this way, the value of the dependent variable can be “predicted” for values of the independent variable. That is, if a book is to contain 350 pages, the equation derived from the data in Figure 11-27 could be used to predict the price of the book. The fitting of an equation to experimental data is frequently referred to as *regression*.

The simplest approach is to fit the data to a straight line using the equation

$$\hat{Y} = a + bX$$

where

$\hat{Y}$ (pronounced “y-hat”) is the predicted value of the dependent variable Y

X is the independent variable

a and b are constants to be determined from the experimental data

This equation is linear, and the approach used to determine the constants (a and b) is called *linear regression*.

For terminology, suppose our data consist of N observations. For the i th observation, the value of the dependent variable is Y_i and the value of the independent variable is X_i . Similarly, we shall let $\bar{X}$ and $\bar{Y}$ (pronounced “y-bar”) be the respective means (arithmetic averages), defined as follows:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i = \frac{1}{N} (X_1 + X_2 + \cdots + X_{N-1} + X_N)$$

$$\bar{Y} = \frac{1}{N} \sum_{i=1}^N Y_i = \frac{1}{N} (Y_1 + Y_2 + \cdots + Y_{N-1} + Y_N)$$

For the data in Figure 11-27, the values of $\bar{X}$ and $\bar{Y}$ are 347 and \$12.12, respectively.

The deviations are the differences between the experimental observations of the dependent variable Y and the corresponding predicted value $\hat{Y}$ for the dependent variable. That is, for the i th data point, the deviation is defined as

$$d_i = Y_i - \hat{Y}_i = Y_i - (a + bX_i)$$

Figure 11-28 illustrates the deviations for a specific case.

In performing the linear regression, the objective is to determine the values of a and b that minimize the sum of the squares of the deviations (often referred to

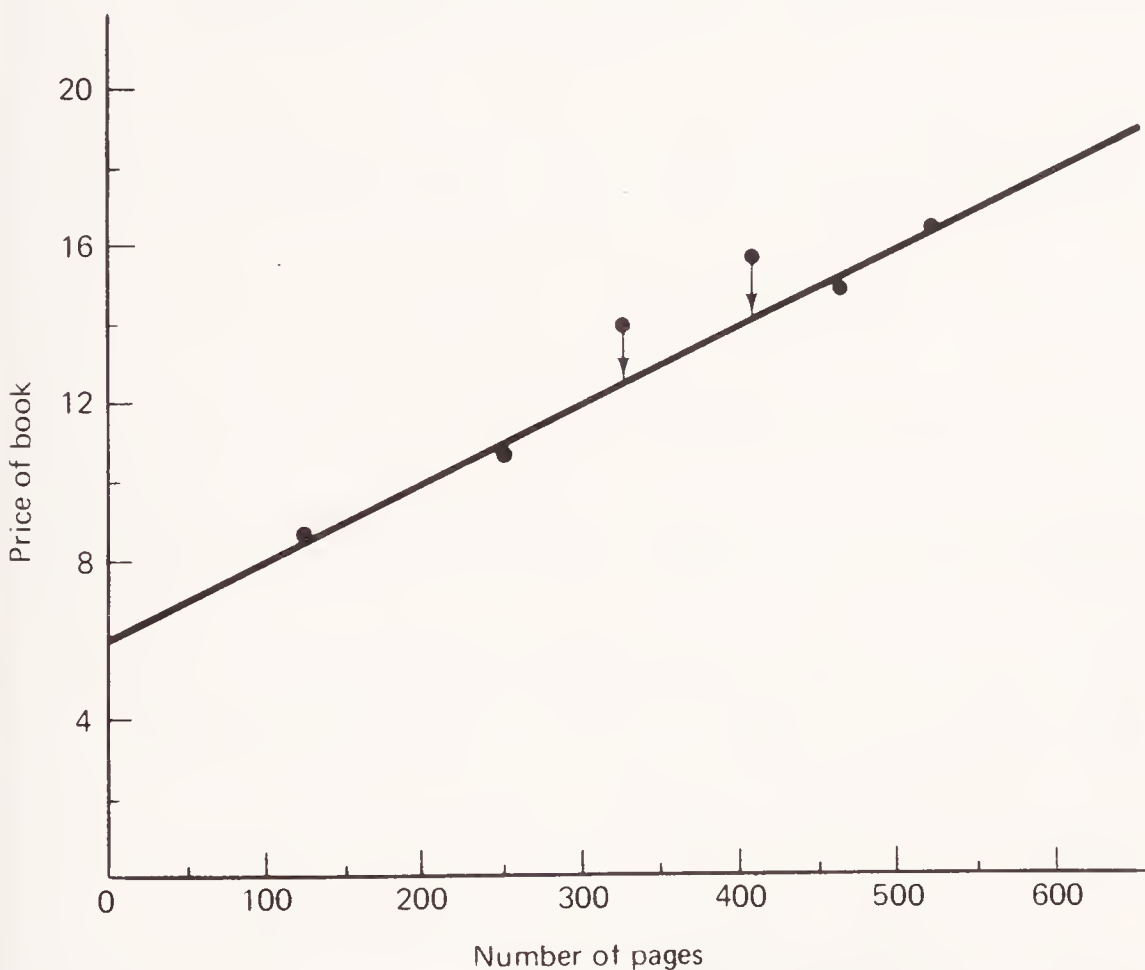


Figure 11-28 Deviations for the case in which $a = 6$ and $b = 0.02$

as the “residual sum of squares”). Mathematically, this can be represented as follows:

$$\begin{aligned}\min_{a,b} \sum_{i=1}^N d_i^2 &= \min_{a,b} \sum_{i=1}^N (Y_i - \hat{Y}_i)^2 \\ &= \min_{a,b} \sum_{i=1}^N (Y_i - a - bX_i)^2\end{aligned}$$

For students who have completed calculus, this minimization is quite straightforward, and upon carrying out the necessary operations, equations are obtained for a and b . Alternatively, the results can be derived without the use of calculus, but the process is too long and involved for a text of this type.

Instead of presenting only the final equations, we shall use an “it can be shown that” or two so that the gist of the derivation can be appreciated. First, the final equation will pass through the point $(\bar{X}, \bar{Y})$, or, in other words, the constants a and b must satisfy the equation

$$\bar{Y} = a + b\bar{X}$$

Since $\bar{X}$ and $\bar{Y}$ are computed directly from the data, this permits a to be expressed in terms of b :

$$a = \bar{Y} - b\bar{X}$$

Substituting into the expression for the residual sum of squares yields

$$\begin{aligned}\sum_{i=1}^N d_i^2 &= \sum_{i=1}^N (Y_i - \hat{Y}_i)^2 \\ &= \sum_{i=1}^N (Y_i - a - bX_i)^2 \\ &= \sum_{i=1}^N [Y_i - (\bar{Y} - b\bar{X}) - bX_i]^2 \\ &= \sum_{i=1}^N [(Y_i - \bar{Y}) - b(X_i - \bar{X})]^2 \\ &= \sum_{i=1}^N (Y_i - \bar{Y})^2 - 2b \sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y}) + b^2 \sum_{i=1}^N (X_i - \bar{X})^2\end{aligned}$$

The summation terms

$$\sum_{i=1}^N (Y_i - \bar{Y})^2$$

$$\sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y})$$

$$\sum_{i=1}^N (X_i - \bar{X})^2$$

can be computed directly from the original data and in no way depend upon the values of a or b .

If we let these terms be denoted as S_{yy} , S_{xy} , and S_{xx} , respectively, the equation for the residual sum of squares becomes

$$\sum_{i=1}^N d_i^2 = S_{yy} - 2bS_{xy} + b^2S_{xx}$$

Recall that the residual sum of squares is to be minimized; that is,

$$\min_{a,b} \sum_{i=1}^N d_i^2 = \min_b (S_{yy} - 2bS_{xy} + b^2S_{xx})$$

Since S_{yy} is a constant, it does not affect the value of b that minimizes the residual sum of squares, and the above expression becomes

$$\min_b (b^2S_{xx} - 2bS_{xy})$$

A student of calculus could readily show that this minimum occurs when

$$b = \frac{S_{xy}}{S_{xx}}$$

Substituting into the previous equation for the residual sum of squares gives

$$\begin{aligned} \sum_{i=1}^N d_i^2 &= S_{yy} - 2bS_{xy} + b^2S_{xx} \\ &= S_{yy} - \frac{2S_{xy}^2}{S_{xx}} + \frac{S_{xy}^2}{S_{xx}} \\ &= S_{yy} - \left(\frac{S_{xy}}{S_{xx}} \right) S_{xy} = S_{yy} - bS_{xy} \end{aligned}$$

Therefore, the final equations are as follows:

$$b = \frac{\sum_{i=1}^N (Y_i - \bar{Y})(X_i - \bar{X})}{\sum_{i=1}^N (X_i - \bar{X})^2}$$

$$a = \bar{Y} - b\bar{X}$$

$$\sum_{i=1}^N d_i^2 = \sum_{i=1}^N (Y_i - \bar{Y})^2 - b \sum_{i=1}^N (Y_i - \bar{Y})(X_i - \bar{X})$$

In implementing these calculations on a computer, suppose we enter a value of the independent variable X_i and the corresponding value of the dependent variable Y_i . The flowchart in Figure 11-29 performs the necessary operations as follows:

1. The values are read and stored in the single-subscripted arrays X and Y .
2. The means are computed.
3. The sums of squares and cross-products are computed.
4. The constants and residual sum of squares are computed.

Reading the value of N permits the flowchart in Figure 11-29 to be independent of the number of observations in the data set.

One major disadvantage of the flowchart in Figure 11-26 is that the observations must be stored in arrays, which is not always convenient for large data sets. This can be avoided by using alternative expressions for the summation terms. For example, suppose we expand the following summation expression:

$$\sum_{i=1}^N (X_i - \bar{X})^2 = \sum_{i=1}^N X_i^2 - 2 \sum_{i=1}^N X_i \bar{X} + \sum_{i=1}^N \bar{X}^2$$

Since $\bar{X}$ is independent of i ,

$$\sum_{i=1}^N X_i \bar{X} = \bar{X} \sum_{i=1}^N X_i = N\bar{X}^2 \quad \text{since} \quad \sum_{i=1}^N X_i = N\bar{X}$$

$$\sum_{i=1}^N \bar{X}^2 = N\bar{X}^2$$

The final result is

$$\sum_{i=1}^N (X_i - \bar{X})^2 = \sum_{i=1}^N X_i^2 - N\bar{X}^2$$

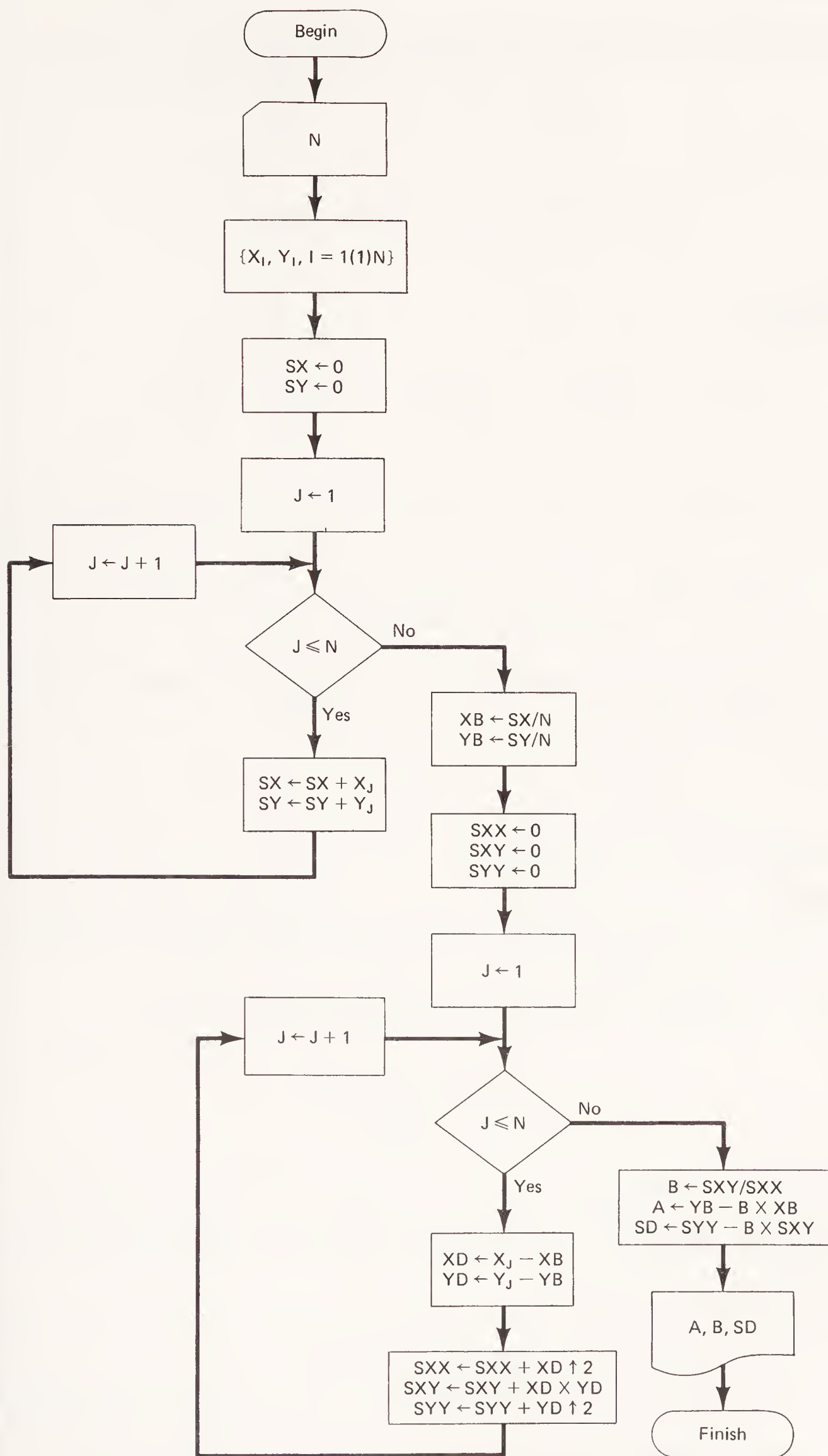


Figure 11-29 Flowchart for performing linear regression

402 Similar results can be obtained for the other summation terms:

$$\sum_{i=1}^N (Y_i - \bar{Y})^2 = \sum_{i=1}^N Y_i^2 - N\bar{Y}^2$$

$$\sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y}) = \sum_{i=1}^N X_i Y_i - N\bar{X}\bar{Y}$$

The summations

$$\sum_{i=1}^N Y_i^2 \quad \sum_{i=1}^N X_i^2 \quad \sum_{i=1}^N X_i Y_i \quad \sum_{i=1}^N X_i \quad \sum_{i=1}^N Y_i$$

can now be computed as the observations are read and the values of $\bar{X}$, $\bar{Y}$,

$$\sum_{i=1}^N (X_i - \bar{X})^2 \quad \sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y}) \quad \sum_{i=1}^N (Y_i - \bar{Y})^2$$

computed subsequently. As illustrated by the flowchart in Figure 11-30, no arrays are needed.

EXERCISES

- 11-1. Develop a flowchart analogous to the one in Figure 11-2 for computing e^x by summing the series given in Section 11.2.
- 11-2. The function $\sin^2 x$ can be represented by the following series:

$$\sin^2 x = \sum_{n=1}^{\infty} \frac{(-1)^{n+1} 2^{2n-1} x^{2n}}{(2n)!}$$

- Develop a flowchart analogous to the one in Figure 11-2 for computing $\sin^2 x$.
- 11-3. On the interval $0 < x \leq 2$, the function $\log x$ can be represented by the following series:

$$\log x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} (x-1)^n}{2.303 n}$$

- Develop a flowchart analogous to the one in Figure 11-2 for computing $\sin^2 x$.
- 11-4. A number of interesting exercises can be devised around the effect of

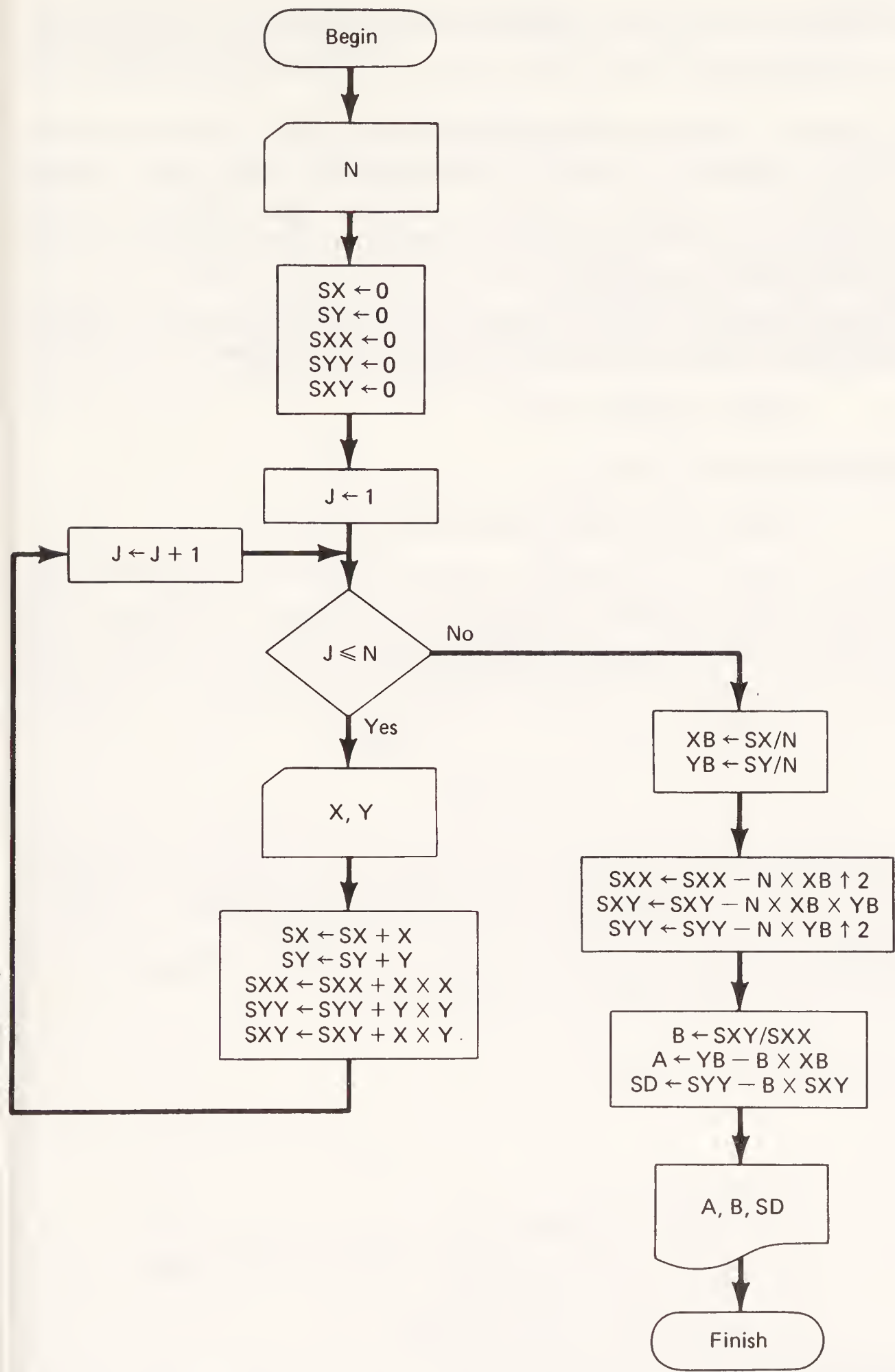


Figure 11-30 Flowchart for performing linear regression without using arrays

computational precision upon numerical techniques. For example, the series e^x is represented as follows:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

404

where the n th term is $x^n/n!$. As pointed out in Exercise 11-1, an alternative approach to computing the terms in this series is to multiply the preceding term by x/i .

To determine the effect of computational precision, let's first write a function to limit the precision of a number to three significant digits, using rounding for the last digit. For example, 127568.2 would become 128000. to three significant digits; 0.0012743 would become 0.00127 to three significant digits. Logarithms can be used for this purpose.

In evaluating the above series, the function, say ROUND, should be inserted after every addition, subtraction, etc. For example, the statement

$$\text{TERM} = \text{TERM} \times X / I$$

should be programmed as follows:

$$\text{TERM} = \text{ROUND} (\text{ROUND} (\text{TERM} \times X) / I)$$

Compute $e^{-5.5}$ directly according to the above series using five-digit precision. Terminate the computation of the series whenever an additional term changes the value of the summation by less than 0.001 percent. Compare with the value of $e^{-5.5}$ computed by the EXP function. Then compute $e^{-5.5}$ by noting that

$$e^{-5.5} = 1/e^{5.5}$$

and evaluate $e^{5.5}$ by the above series. Compare with the value of $e^{-5.5}$ computed in this manner to those obtained above, as was done in Table 11-1.

11-5. Equations of state are used to relate the pressure, temperature, and volume of a gas. One such equation is Van der Waal's:

$$(P + a/V^2) (V - b) = RT$$

where

P = pressure, atm

T = absolute temperature, °K

R = universal gas law constant, 0.08205 atm/°K mole

V = molar volume, liters/mole

a, b = constants for a particular gas

For nitrogen, the values for the constants are

$$a = 1.390 \frac{(\text{liters})^2 \times \text{atm}}{(\text{mole})^2}$$

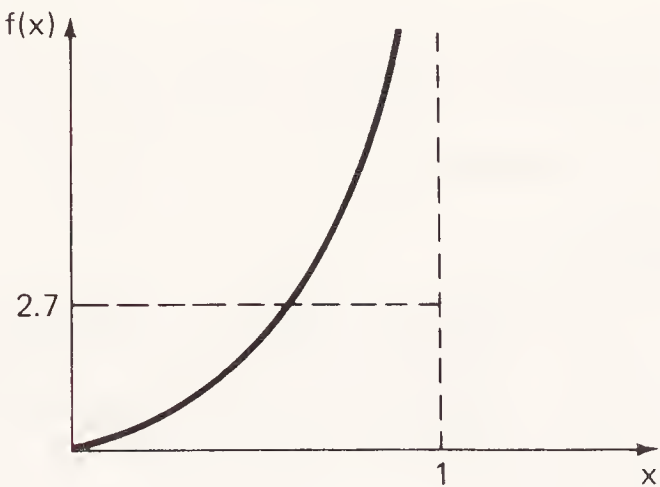
$$b = 0.03913 \text{ liter/mole}$$

Using interval halving, compute the volume of a mole of nitrogen at 300°K and 1.2 atm pressure. This volume is less than 50 liters.

11-6. In the study of chemical equilibria, the solution of equations of the following type often occur:

$$\frac{x(4+x)(2-x)}{(1-x)^2} = 2.7$$

At $x = 0$, the left side of the equation equals zero. At $x = 1$, the denominator is 0, and therefore the left side equals ∞ . Therefore, at some value of x between 0 and 1, the left-hand side must equal 2.7, and we have a solution to the equation (see accompanying figure). Prepare the flowchart for a program using interval halving to determine the solution.

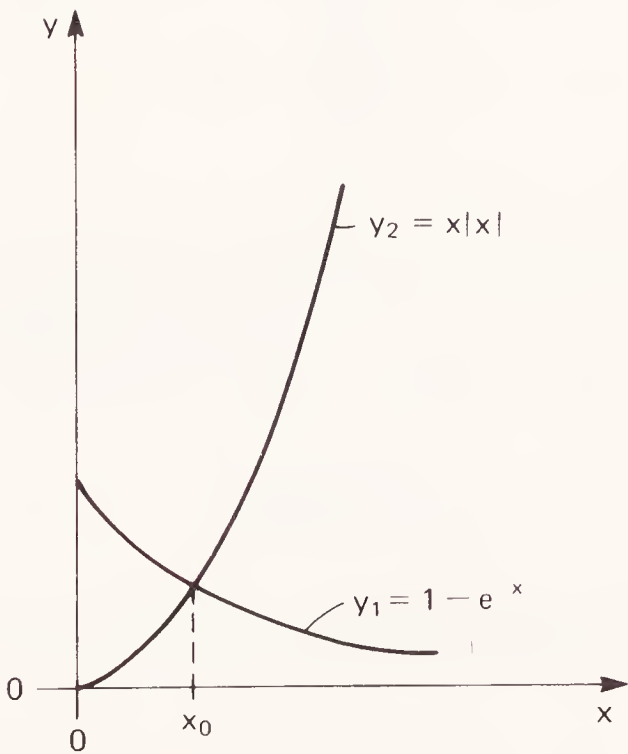


11-7. The equation in the previous exercise also has a solution somewhere between $x = 1$ and $x = 2$. Prepare the flowchart for a program using interval halving to determine this solution to the equation.

11-8. Suppose the following equations are to be solved for x and y :

$$y = 1 - e^{-x}$$

$$y = x |x|$$



406

As we have two equations in two unknowns, they can be solved for x and y . Graphically, this is the problem of determining the intersection of the two curves in the accompanying illustration. Although one could equate the two equations to obtain one equation in one unknown, the result would be nonlinear and no advantage is accrued. Given that a solution for these two equations exists somewhere between $x = 0$ and $x = 5$, devise a scheme using interval halving to determine the solution.

11-9. Suppose we change the equations in the above exercise to the following

$$x = 1 - e^{-y} \tag{1}$$

$$y = x |x| + 1 \tag{2}$$

Although certain rearrangements can be made so that interval halving can be applied to these equations, this is not always convenient. Let's explore another approach, specifically the following:

- 1. Assume a value for x .
- 2. Using this value of x , compute a value for y from Equation 2.
- 3. Using this value of y , compute a value for x from Equation 1.
- 4. Using the value of x computed in step 3 as the new assumed value for x , repeat from step 1 until the value assumed for x in step 1 equals the value computed in step 3.

Known as *direct substitution*, this method generates the following sequence of values for this example:

Value for x assumed in (1)	Value for y computed in (2)	Value for x computed in (3)
5.000	26.000	1.000
1.000	2.000	0.865
0.865	1.748	0.826
0.826	1.682	0.814
0.814	1.663	0.810
0.810	1.657	0.809
0.809	1.655	0.809

Prepare the flowchart for a program that embodies this approach.

11-10. In effect, the above exercise treated the solution of equations of the type

$$x = f(x)$$

For the above exercise, the equation was

$$x = 1 - e^{-(x|x|+1)}$$

The direct substitution method reduces to the repetitive computation of the recursive equation

$$x_{i+1} = f(x_i)$$

where x_0 must be assumed.

Naturally, it is not difficult to devise situations where this approach does not converge to an answer. In these cases, the recursive equation can be modified to the following:

$$x_{i+1} = x_i + k(f(x_i) - x_i)$$

where k is an arbitrary chosen constant. When k equals 1, this equation reduces to the one given earlier. In effect, the term $f(x_i) - x_i$ is the error in the equation for the assumed x_i , and the recursive equation causes the assumed value of x to be changed by an amount equal to a constant times this error.

The best value of k is the one that leads to the correct answer in the minimum number of iterations on the recursive equation. Unfortunately, the selection of k is a trial-and-error procedure, and cases can be devised where the best value of k is a fraction less than 1, a positive number greater than one, or even a negative constant.

Prepare a flowchart using this method to solve the equation

$$x = (x + 0.05)^{0.95}$$

If a computer is available, program this exercise and determine the best value of k .

11-11. Same as the above exercise but solve the equation

$$x = (x - 0.05)^{1.05}$$

for $x > 0.05$.

11-12. The substitution method can also be applied to equations of the following type that involve two unknowns:

$$x = f_1(x, y)$$

$$y = f_2(x, y)$$

The recursive equations are as follows:

$$x_{i+1} = x_i + k_1 (f_1(x_i, y_i) - x_i)$$

$$y_{i+1} = y_i + k_2 (f_2(x_i, y_i) - y_i)$$

where values for x_0 and y_0 must be assumed.

Prepare the flowchart for a program to solve the following equations for x and y :

$$x = 4 + x^{0.22} \sqrt{xy}$$

$$y = x^2 (1 - y^{0.29})$$

Use the value 0.05 for k_1 and k_2 , but do not print after every iteration.

11-13. Suppose we would like to design a cylindrical case of volume V that can be constructed using the least amount of metal. The surface area of a cylinder of volume V is a function of the radius r as given by the following equation:

$$A = \frac{2V}{r} + 2\pi r^2$$

Use the golden section to find the value of r that minimizes A if V is 2.6 ft^3 . It is believed that r will be less than 2 feet.

11-14. Suppose we would like to find the point on the graph of the equation

$$y = x^3$$

that is closest to the point $x = 4, y = 1$. This distance as a function of x is given by the equation

$$d = \sqrt{(x - 4)^2 + (x^3 - 1)^2}$$

Using the golden section, determine the value of x that minimizes d . It is suspected that x will be less than 4.

11-15. An interesting phenomenon occurring in heat transfer is that addition of insulation to the surface of a cylindrical conductor such as the wire illustrated in the accompanying figure does not always reduce the heat loss. The equation for the heat loss is as follows:

$$q = \frac{2\pi k (T_i - T_a)}{\ln(r_o/r_i) + k/hr_o}$$

where k = thermal conductivity of insulation material, Btu hr ft $^{\circ}$ F

T_i = temperature at surface of conductor, $^{\circ}$ F

T_a = temperature of the surrounding air, $^{\circ}$ F

r_o = outer radius of insulation, feet

r_i = inner radius of insulation, feet

h = heat transfer coefficient between insulation in air, Btu/hr ft ^{2} $^{\circ}$ F

q = heat loss, Btu/hr per foot of cable

Using the golden section, determine the value of r_o that maximizes q under the following conditions:

$$k = 0.1 \text{ Btu/hr ft } ^{\circ}\text{F}$$

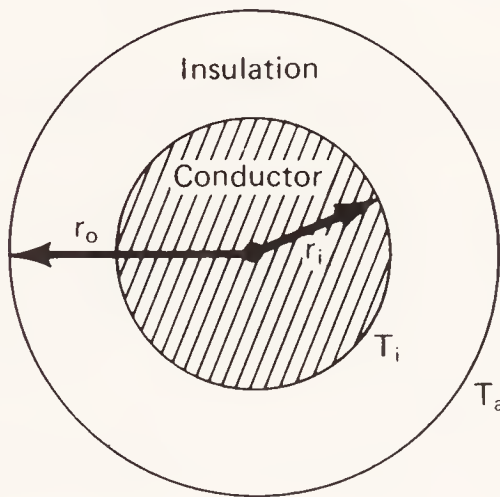
$$h = 1.5 \text{ Btu/hr ft}^2 \text{ } ^{\circ}\text{F}$$

$$r_i = \frac{1}{4} \text{ in} = \frac{1}{48} \text{ foot}$$

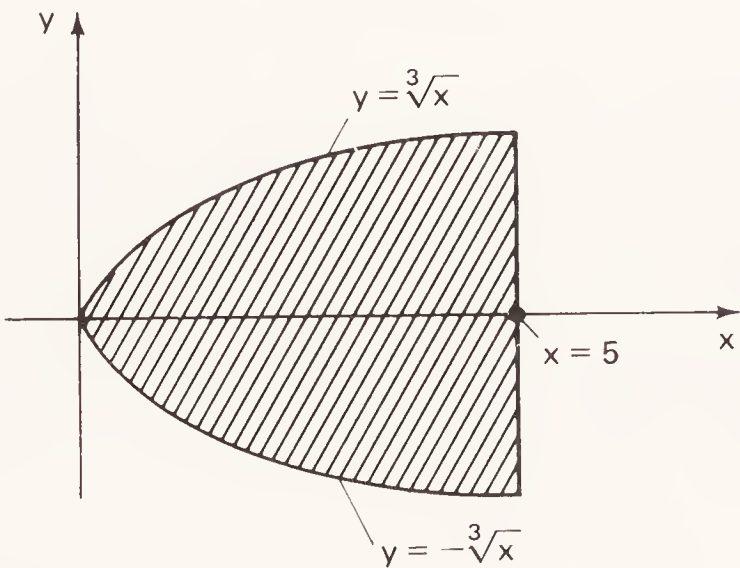
$$T_i = 150^{\circ}\text{F}$$

$$T_a = 80^{\circ}\text{F}$$

The maximum value is believed to be less than 3 feet.



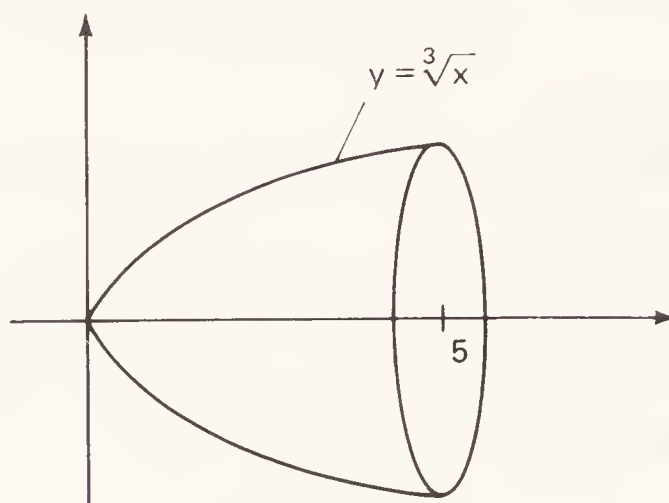
11-16. A plane area is constructed as shown in the accompanying figure. Using the numerical integration techniques presented in this chapter, compute the area of the cross-hatched region.



410 11-17. One way to generate solid figures is to take a graph such as

$$y = \sqrt[3]{x}$$

and rotate it around the x axis as illustrated in the accompanying figure. Using the integration techniques presented in this chapter, compute the surface area of the object in the accompanying illustration. *Note:* The circumference of the object as a function of x is $2\pi\sqrt[3]{x}$.



11-18. Same as the above exercise but compute the volume of the object. *Note:* The cross-sectional area of the object as a function of x is $\pi x^{2/3}$.

11-19. The Gauss iteration method is an iterative approach for solving a set of simultaneous linear equations that can be applied whenever one of the coefficients of a different root in each equation is large compared with the others. This large coefficient can always be put in the diagonal or a_{ii} position by rearranging the rows and columns. If we let $a'_{ij} = a_{ij}/a_{ii}$ and $b'_i = b_i/a_{ii}$, the 3×3 set of equations in Section 11.6 can be expressed as follows:

$$\begin{aligned}x_1 &= b'_1 - a'_{12}x_2 - a'_{13}x_3 \\x_2 &= b'_2 - a'_{21}x_1 - a'_{23}x_3 \\x_3 &= b'_3 - a'_{31}x_1 - a'_{32}x_2\end{aligned}$$

Starting with assumed values for x_1 , x_2 , and x_3 , they may be substituted into the right side of these equations, and new values for x_1 , x_2 , and x_3 computed. These values may now be used to compute the right side, and this process may be continued until the computed values equal the assumed values.

It can be shown that this procedure will converge to the solution provided that the sum of the absolute values of the a' coefficients in each equation is less than or equal to one, with the "less than" holding for at least one equation. That is,

$$\sum_{\substack{j=1 \\ j \neq i}}^N |a_{ij}| \leq 1 \quad (i = 1, 2, \dots, n)$$

where the $\leq$ holds for at least one i .

Prepare the flowchart for a program that reads a set of equations, computes the a' coefficients, and carries out the iterations. This method should function properly for the following set of equations:

$$11x_1 + 6x_2 - 3x_3 = 4$$

$$4x_1 + 10x_2 - 2x_3 = 3$$

$$x_1 - 5x_2 + 9x_3 = 1$$



ASSEMBLY LANGUAGE PROGRAMMING

The purpose of this Appendix is to introduce the student to assembly-language programming. This book will take a somewhat different approach to this subject than most others. The usual approach is to introduce a contrived set of instructions that comprise an artificial assembly language that is given a catchy name. Our approach will be to introduce subsets of two regular assembly languages used on commercial computers. In this way, the introduction here is to a language that is used commercially instead of an artificial substitute.

In view of the fact that this is to be only an introduction, we shall place certain restrictions on our activities. These are

1. Except for the sections on indexed addressing, only the storage locations available in the form of registers in the CPU will be utilized. Because of this, we shall be restricted to sixteen storage locations or fewer, but these are sufficient to solve some complex problems.
2. We shall not consider instructions for floating-point operations.
3. One of the most difficult aspects of most assembly languages is input/output. To avoid this, we shall introduce either macros or subroutine calls for input/output. A *macro* is a pseudo-assembly language instruction that actually stands for several instructions which the assembler inserts in its place.

In this appendix, we shall introduce a subset of the instructions for the IBM/360 Series and a subset of the instructions for the XDS Sigma 5/7 series. Similar subsets for other machines can be readily developed. Our objective is to introduce only the minimum number of instructions sufficient to solve some reasonable problems. In many cases, other instructions could be used to accomplish certain operations more efficiently, but we shall refrain from introducing these in order to keep our “instruction set” short.

Both the System 360 and the XDS Sigma 5/7 series are basically byte-oriented machines. Both have eight-bit bytes, four of which comprise a word totaling thirty-two bits. Each machine has sixteen registers, each containing thirty-two bits, or the equivalent of one word. All the arithmetic instructions that we shall introduce will operate on a word of information. Except for a few restrictions, which we shall point out as they apply to our instructions, all of the registers are equivalent. For example, addition can be performed in any of the registers. In numbering the registers, we begin with 0 and continue through 15.

To use the input/output routines described later in this section, certain registers should not be used. On the System 360, registers 1, 13, 14, and 15 should not be used in the remainder of the program, leaving twelve free registers. On the Sigma Series, register 15 should not be used, leaving fifteen free registers.

**A.1. ARITHMETIC INSTRUCTIONS
FOR THE SYSTEM 360**

The following is a subset of the arithmetic instructions for performing integer arithmetic on the IBM/360:

Load register LR *i,j*

This instruction moves the contents from register *j* to register *i*. The contents of register *j* are not changed. Although *i* may equal *j*, this instruction would accomplish nothing.

Add register AR *i,j*

This instruction adds the contents of register *j* to the contents of register *i*. The contents of register *j* are not changed; the result appears in register *i*. If *i* equals *j*, the result is equivalent to multiplication by 2.

Subtract register SR *i,j*

This instruction subtracts the contents of register *j* from the contents of register *i*. The contents of register *j* are not changed; the result appears in register *i*. If *i* equals *j*, the result is zero.

Load L *i, = F 'number'*

This instruction inserts the value of the integer *number* into register *i*.

**A.2. INPUT/OUTPUT OPERATIONS
FOR THE SYSTEM 360**

The following macros are used for input and output:

Write word WWD *i*

The contents of register i are written on the output unit in integer format. The contents of register i are not changed, but this instruction is subject to the restriction that i may not equal 0. Furthermore, the contents of register 1 are altered by this instruction.

Read word RWD i

The next record in the input unit is scanned, and the first integer number found in this record is loaded into register i . This instruction is subject to the restriction that i may not equal 0. Furthermore, the contents of register 1 are altered by this instruction.

A.3. BRANCH INSTRUCTIONS

Using the IBM/360 assembler, a label may be attached to any of the instructions that we shall present. This label is restricted to eight characters, the first of which must be alphabetic. Branch instructions may cause the flow of control to jump (forward or backward) to the instruction with the referenced label.

The following branch instructions are available:

Branch unconditionally B *label*

The next instruction to be executed is the one to which *label* is attached.

Branch on minus BM *label*

In the conventional computer with a single accumulator, this instruction would cause control to transfer to the instruction to which *label* is attached only if the contents of the accumulator were negative. On computers with multiple registers, however, this instruction operates differently. On such computers, a special set of bits known as the *condition codes* govern most conditional transfer operations. Part of the function of most arithmetic instructions, including AR and SR, is to set the condition codes. For example, if the result generated by an SR instruction is negative, the condition codes are set accordingly.

When a BM instruction is executed, the transfer occurs only if the current status of the condition codes is negative.

The BP (Branch on Positive) and BZ (Branch on Zero) instructions are also recognized.

The LR instruction does not alter the condition codes. However, the “load and test register” instruction

LTR i, j

does set the condition codes. In all other respects, LTR is equivalent to LR. Executing an LTR instruction with i equal to j only causes the condition codes to be set.

416 A.4. EXAMPLE

As our first example of a program in this subset of the assembly language, suppose we write a program to compute the sum of five numbers to be read from data cards.

The program to accomplish this is given in Figure A-1. The use of the various registers is as follows:

Register	Use
2	Counter for the number of iterations
3	Constant 1
4	Holds the sum
5	Recipient for values read

In constructing the counter utilizing a BM instruction, one approach is to initialize the counter to -5 and add one to it during each iteration. As long as the counter remains negative, the BM instruction returns control to the top of the loop to begin another iteration. When the counter reaches zero, control passes to the WWD macro following the BM instruction. Note also that the counter is incremented by the AR instruction immediately prior to the BM instruction, thereby insuring that the condition codes are set properly. There are, of course, other ways of constructing the loop, but this is an efficient method.

As our second example, we shall compute the nineteenth term in the Fibonacci sequence introduced in Chapter 3. Recall that this sequence is specified as follows:

Term 1 = 0

Term 2 = 1

Term *i* = Term (*i* - 1) + Term (*i* - 2)

That is, the first two terms are defined as 0 and 1. Thereafter, a given term equals the sum of the two previous terms.

The flowchart of an algorithm to compute this series is given in Figure 3-9.

LCCP	L	2,=F'-5'	INITIALIZE COUNTER TO -5
	L	3,=F'1'	STORE CONSTANT 1 IN REG 3
	L	4,=F'0'	INITIALIZE SUM TO ZERO
	RWD	5	READ NUMBER
	AR	4,5	ADD NUMBER TO SUM
	AR	2,3	ADD 1 TO COUNTER
	BM	LCCP	ARE WE THROUGH
	WWD	4	WRITE SUM

Figure A-1 Sum of five numbers

```

L      8.=F*1*      LCAD CONSTANT 1 INTO R8
L      2.=F*0*      INITIALIZE ENT2 TO 0
L      3.=F*1*      INITIALIZE ENT1 TO 1
RWD    5            READ NENT
LCOP   L      6.=F*3*      SET J=3
      LR      7,5
      SR      7,6          J LESS THAN OR EQUAL TO NENT
      RM      OUT
      LR      4,3
      AR      4,2          ENT = ENT1 + ENT2
      LR      2,3          ENT2 = ENT1
      LR      3,4          ENT1 = ENT
      AR      6,8          J = J+1
      B      LOOP
OUT    WWD      5          WRITE NENT
      WWD      3          WRITE ENT
```

Figure A-2 Computation of Fibonacci sequence

The corresponding assembly language program is given in Figure A-2. The allocation of storage to the variables and constants in Figure 3-9 is as follows:

Register	Use
2	Variable ENT2
3	Variable ENT1
4	Variable ENT
5	Variable NENT
6	Variable J
7	For computing NENT – J
8	Constant 1

The explanation of the use of the various instructions is also given in Figure A-2.

A.5. STATEMENT FORMAT AND REMAINDER OF THE PROGRAM

Figure A-3 shows the coding form used for preparing assembly-language programs. Although the statement format need not follow strict rules, general practice is to start the label in column 1, start the instruction in column 9, start the fields in column 16, and start the comments with column 31. If an asterisk appears in column 1, everything else on that card is treated as comments. Caution: If column 72 contains any nonblank character, the *next* line is treated as a continuation of the current line.

Up to this point, we have discussed only the instructions comprising the heart of the program. The first three statements in our assembly language programs for the System 360 must be as follows:

418

MAIN

START

0

PRINT

NOGEN

INITIAL

The last two statements must be

EXIT

END

MAIN

A simple listing of the computer print-out of the program in Figure A-2 for computing the Fibonacci sequence is given in Figure A-4.

A.6. MULTIPLICATION AND DIVISION
ON THE SYSTEM 360†

Multiplication and division are different from other operations for integer arithmetic in that they both involve double-length words (doublewords).

Multiplication

MR

i,j

This instruction multiplies the contents of register $i + 1$ by the contents of register j , the result appearing as a doubleword in registers i and $i + 1$. When two thirty-two-bit numbers are multiplied, the result is sixty-four-bits in length and thus requires two registers for storing. But unless the product is numerically too large to be stored in a thirty-two-bit word, the contents of register $i + 1$ comprise the product desired. The contents of register j are not changed. Register i must be an even-numbered register.

As an example, suppose we want to multiply the quantity in register 3 by the quantity in register 6. Neither of these quantities is to be destroyed during the operation. Assuming registers 10 and 11 are available for use, the appropriate instructions are

LR

11,3

MR

10,6

Register 11 contains the answer.

Division

DR

i,j

The contents of registers i and $i + 1$, treated as a doubleword, are divided by the contents of register j . The quotient appears in register $i + 1$; the remainder appears in register i . In order to obtain a thirty-two-bit quotient when dividing by a thirty-two-bit quantity, the dividend must be sixty-four-bits in length and thus the need for the doubleword. The contents of register j are not altered by this operation. Register i must be an even-numbered register.

As an example of the use of the MR and DR instructions, the program in

†The instructor may prefer to introduce macros for these operations.

* A standard card form, IBM electro 0509, is available for punching source statements from this form. Instructions for using this form are in any IBM System/360 assembler language reference manual. Address comments concerning this form to IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden.

* No of forms per pad may vary slightly.

Figure A-3 Assembly-language coding form

LOC	OBJECT CODE	ADDR1	ADDR2	STMT	SOURCE STATEMENT
000000				1 MAIN	START 0
				2	PRINT NOGEN
				3	INITIAL
000074	5880 D098		000C0	20	L 8,=F.1.
000078	5820 D09C		000C4	21	L 2,=F.0.
00007C	5830 D098		000C0	22	L 3,=F.1.
				23	RWD 5
				27	L 6,=F.3.
00008A	5860 D0A0		000C8	28	LR 7,5
00008E	1875			29	SR 7,6
000090	1876			30	BM OUT
000092	4740 D07C		000A4	31	LR 4,3
000096	1843			32	AR 4,2
000098	1A42			33	LR 2,3
00009A	1823			34	LR 3,4
00009C	1834			35	AR 6,8
00009E	1A68			36	B LOOP
0000A0	47F0 D066		0008E	37	WWD 5
				41	WWD 3
				45	EXIT
				48	END
0000C0	000000001			49	MAIN
0000C4	000000000			50	=F.1.
0000C8	000000003			51	=F.0.
					=F.3.

LOAD CONSTANT 1 INTO R8

INITIALIZE ENT2 TO 0

INITIALIZE ENT1 TO 1

READ NENT

SET J=3

J LESS THAN OR EQUAL TO NENT

ENT = ENT1 + ENT2

ENT2 = ENT1

ENT1 = ENT

J = J+1

WRITE NENT

WRITE ENT

a. Program
Figure A-4 Assembly-language program corresponding to Figure 3-9

19
2584

b. Output

Figure A-4 (Continued)

Figure A-5 corresponds to the prime-number algorithm in Figure 2-15. The allocation of storage is as follows:

Register	Use
2	Variable <i>N</i>
3	Constant 1
4	Number
5	Utility register
8,9	Multiplication and division

The first instruction initializes *N* to 2, and the second instruction places the constant 1 in register 3. The number is then read into 4. The constant 1 is placed in register 9, and the MR instruction is used to multiply by the contents of register 4 (the number). This creates a doubleword in registers 8 and 9. The DR instruction is executed to divide by *N* (register 2), obtaining the quotient in register 9 and the remainder in register 8. To use the BM instruction to determine if the remainder is nonzero, register 5 is set to zero, and the remainder (register 8) subtracted from it. If the result is negative, then *N* does not exactly divide the number. Control transfers to label POS, which increments *N*, com-

```

L      2,=F'2' SET N = 2
L      3,=F'1' PUT CONSTANT 1 IN REG. 3
RWD    4      READ NUMBER
LCOP   LR     9,3  PUT 1 IN REG. 9
MR      8,4     MULTIPLY BY NUMBER TO OBTAIN DOUBLE-WORD
DR      8,2     DIVIDE BY N
L      5,=F'0' PUT 0 IN REG. 5
SR      5,8     SUBTRACT REMAINDER
BM      POS    NEG. REMAINDER INDICATES NON-INTEG. RESULT
WWD     3      WRITE CONSTANT 1 TO INDICATE NOT PRIME
B       OUT    BRANCH TO EXIT
POS     AR     2,3  ADD 1 TO N
        LR     5,2
        SR     5,4  DOES N EQUAL NUMBER
        BM     LOOP
        WWD    4      WRITE NUMBER INDICATING IT IS PRIME
OUT     EXIT
        END     MAIN
```

Figure A-5 Assembly-language program corresponding to Figure 2-6

422 pares with the number, and returns to LOOP if N is less than the number. If the remainder is zero, number must not be prime, so the constant 1 is written and we branch to exit.

A.7. ARITHMETIC INSTRUCTIONS FOR THE SIGMA 5/7 SERIES

The following is a subset of the arithmetic instructions for performing integer arithmetic on the XDS Sigma 5/7 Series:

Load word LW,*i* *j*

This instruction moves the contents from register j to register i . The contents of register j are not changed. This instruction also sets the condition codes on the Sigma series.

Add word AW,*i* *j*

This instruction adds the contents of register j to the contents of register i . The contents of j are not changed; the result appears in register i and the condition codes are set.

Subtract word SW,*i* *j*

This instruction subtracts the contents of register j from the contents of register i . The contents of register j are not changed; the result appears in register i and the condition codes are set.

Load immediate LI, *i* *number*

This instruction inserts the integer *number* into register i .

A.8. INPUT/OUTPUT OPERATIONS

On the Sigma series, we will use subroutine calls to accomplish input and output. For input, the instruction “branch and link” is used to call RWD:

BAL,15 RWD

A card is read and scanned to locate a single integer number, and its value is stored in register 0. The condition codes are set according to the sign of the value entered.

For output, the subroutine WWD is called by the instruction

BAL,15 WWD

The routine writes the value contained in register 0 on the output printer. The contents of register 0 are not altered, nor are the condition codes changed.

A.9. BRANCH INSTRUCTION
ON THE SIGMA 5/7 SERIES

Using the Sigma series assembler, a label may be attached to any of the instructions that we shall present. This label is restricted to eight characters, one of which must be alphabetic, and must start in column 1.

The following branch instructions are available:

Branch unconditionally B *label*

The next instruction to be executed is the one to which *label* is attached.

Branch on less than BL *label*

The branch to the instruction to which *label* is attached occurs only if the condition codes are set negative, i.e., if the result of the last arithmetic operation was negative.

Branch on greater than BG *label*

The branch to the instruction to which *label* is attached occurs only if the codes are set positive, i.e., if the result of the last arithmetic operation was positive.

Branch on equal BE *label*

Similar to BL and BG except that the branch occurs if the result of the last arithmetic operation was zero.

A.10. EXAMPLES

As our first example, we shall again write a program to compute the sum of five numbers to be read from data cards. The program to accomplish this is given in Figure A-6.

```

      REF      RWD
      REF      WWD
MAIN  LI,2      -5      COUNTER FOR ITERATIONS
      LI,3      1      CONSTANT 1
      LI,4      0      INITIALIZE SUM TO ZERO
LCOP  BAL,15    RWD     READ A NUMBER
      AW,4      0      ADD TO SUM
      AW,2      3      INCREMENT COUNTER
      BL       LCOP    TEST
      LW,0      4      MOVE SUM TO REG. 0
      BAL,15    WWD     WRITE THE SUM
      CAL1,9    1
      END      MAIN
```

Figure A-6 Program to sum five numbers

424 In our programs for the Sigma 5/7 series, the first two statements must be

```
REF      RWD
REF      WWD
```

These statements define RWD and WWD as subprogram names. The last two statements must be

```
CAL1,9    1
END      label
```

The CAL instruction is equivalent to a “stop” instruction in that it returns control to the executive. The label on the END statement must also be attached to the first statement following the REFs. In Figure A-6, MAIN is used as this label.

In Figure A-6, the use of the registers is as follows:

Register	Use
0	Read and Write
2	Counter
3	Constant 1
4	SUM

The first three statements following the REFs initialize the counter to -5, insert the constant 1 in register 3, and initialize the SUM to zero. A value is then read into register 0 by RWD, and this value is added to the SUM by the first AW instruction. The next AW instruction increments the counter by 1, and then we branch to LOOP as long as the counter is negative. After processing five numbers, the value of SUM is transferred to register 0 and written by WWD.

As the second example of the use of the Sigma series assembly language, Figure A-7 gives the program for computing the Fibonacci number via the approach outlined in Figure 3-9. Use of the registers is as follows:

Register	Use
0	Variable NENT
2	Variable ENT2
3	Variable ENT1
4	Variable ENT
6	Variable J
7	For computing NENT-J
8	Constant 1

The program in Figure A-7 first initializes register 8 to the value 1, initializes ENT2 (register 2) to 0, and initializes ENT1 (register 3) to 1. NENT is then read into register 0 by RWD. The next instruction initializes J (register 6) to 3. Upon entering the loop, J is compared with NENT, and the loop is exited if J exceeds


```

      REF      RWD
      REF      WWD
MAIN  LI,8      1      CONSTANT 1 INTO REG. 8
      LI,2      0      INITIALIZE ENT2 TO 0
      LI,3      1      INITIALIZE ENT1 TO 1
      EAL,15    RWD    READ NENT
      LI,6      3      SET J = 3
LCOP  LW,7      0
      SW,7      6      J LESS THAN OR EQUAL TO NENT
      EL        CUT
      LW,4      3
      AW,4      2      ENT=ENT1 + ENT2
      LW,2      3      ENT2=ENT1
      LW,3      4      ENT1=ENT
      AW,6      8      J=J+1
      B         LCOP
OUT   BAL,15    WWD    WRITE NENT
      LW,J      3      MOVE ENT TO REG. 0
      BAL,15    WWD    WRITE ENT
      CAL1,9    1
      END      MAIN
```

Figure A-7 Program to compute Fibonacci numbers

NENT. A new value of ENT is calculated, and ENT1 and ENT2 are updated. After incrementing J, we return to the top of the loop. After exiting from the loop, the values of NENT and ENT are written.

A.11. MULTIPLICATION AND DIVISION ON THE SIGMA SERIES

Multiplication and division on the Sigma series may or may not involve a doubleword.

Multiply word MW,i j

If *i* is *even*, the product of register *i* + 1 (thirty-two-bits) and register *j* (thirty-two-bits) is a doubleword (sixty-four-bits) and appears in registers *i* and *i* + 1. Unless the product is too large to be represented by a thirty-two-bit word, register *i* + 1 contains the desired result.

If *i* is *odd*, the MW instruction multiplies register *i* by register *j*, the result appearing as a single word in register *i* only.

In neither case are the contents of register *j* changed.

Divide word DW,i j

If *i* is *even*, the contents of the doubleword in registers *i* and *i* + 1 (sixty-four-bits) is divided by the contents of register *j* to obtain the quotient (thirty-two-bits) in register *i* + 1 and the remainder (thirty-two-bits) in register *i*.

If *i* is *odd*, the contents of register *i* are divided by the contents of register *j*, the quotient appearing in register *i*. The remainder is lost.

In neither case is register *j* altered.

As an example of the use of the MW and DW instructions, the prime-number algorithm from Figure 2-15 is shown in assembly language in Figure A-8. The first two instructions set *N* equal to 2 and put the constant 1 into register 3. NUMBER is read into register 0 by RWD. The first three statements in the loop place 1 into register 1, multiply by NUMBER to obtain a doubleword product, and then divide by *N*. The remainder appears in register 8. The next instructions subtract the remainder from zero, a negative result indicating a nonzero remainder. If the remainder is zero, the constant 1 is written to indicate that NUMBER was not prime. If the remainder was nonzero, *N* is incremented by the AW instruction, compared to NUMBER and the loop repeated if *N* is less than NUMBER. If no divisors are found, NUMBER is written indicating that it is a prime number.

A.12. INDEXED ADDRESSING
ON THE SYSTEM 360

Since indexed addressing applies only to addresses in memory, our discussion must begin with the 360's addressing system. The basic unit of storage in the memory on the 360 is a byte, and *all* addresses on the 360 are byte addresses. However, all of the instructions that we shall introduce will manipulate words instead of bytes. On the 360, a word consists of four consecutive bytes with the restriction that the address of the first byte in the word be an integral multiple of 4. This arrangement is illustrated in Table A-1.

On the 360, the address of a word is the address of the first byte; that is, all addresses are actually byte address. Any byte address that is an integral multiple of 4 is a valid word address. For example, the address of the first word in Table A-1 is 8540, the address of the second word is 8544, and the address of the third is 8548.

Table A-1 Arrangement of Memory

VCT	Byte	8540	Word
	Byte	8541	
	Byte	8542	
	Byte	8543	
	Byte	8544	Word
	Byte	8545	
	Byte	8546	
	Byte	8547	
	Byte	8548	Word
	Byte	8549	
	Byte	8550	
	Byte	8551	

In order to use storage locations in memory, the necessary area must be reserved. This is accomplished via the “declare storage” statement, whose format for reserving words in memory is as follows:

```
label          DS      nF
```

where *n* is the number of words to be reserved. The address of the first byte in the area reserved is associated with the label. For example, the following statement would reserve one word in memory:

```
VCT          DS      1F
```

If the reserved area is as illustrated in Table A-1, the address 8540 would be associated with the label VCT. The “F” in this statement specifies a “fullword,” and insures that the address of the first byte in the area reserved will be an integral multiple of 4 and thus qualify as a word address.

With an area in memory reserved via the above DS statement, the contents of a register can be copied into a word in memory via the “store” instruction:

```
ST      n, label
```

when *n* is the number of the register and label designates the address in memory at which the information is to be copied. For example, the instruction

```
ST      3,VCT
```

would copy the contents of register 3 into the word reserved by the statement

```
VCT      DS      1F
```

Since the register on the 360 contains thirty-two-bits and each byte contains eight bits, all of the information in the register can be stored in memory. The ST instruction processes a word, and therefore the single instruction copies the contents of the register into the four consecutive bytes, the first of whose address is associated with label VCT.

To copy the contents of a word in memory into one of the registers, the appropriate instruction is the “load” instruction, whose format is as follows:

```
L      n, label
```

where *n* is the register and label designates the address of the word in memory. For example the instruction

```
L      5, VCT
```

copies the contents of the word reserved by the statement

```
VCT      DS      1F
```

into register 5. Like the ST instruction, the L instruction transfers a word.

On the System 360, any register except register 0 can be used as an index register. To illustrate indexed addressing, suppose the three words in memory reserved by the statement

VCT DS 3F

are as shown in Table A-1. To designate indexed addressing, the number of the register enclosed in parentheses follows the label. For example, the use of register 2 as an index register is designated by

VCT(2)

This is interpreted as “add the contents of register 2 to the address associated with label VCT to obtain the address to be used in executing the instruction in which VCT(2) appears.”

Indexed addressing may be used with either the L or the ST instruction. Consider the following example:

L 5,VCT(3)

If the contents of register 3 are zero, the contents of the first word illustrated in Table A-1 are copied into register 5. If the contents of register 3 is 4, the contents of the second word are copied into register 5. If register 3 contains 8, the contents of the third word are copied into register 5.

In effect, we proceed from one word of memory to the next by incrementing the address by 4. This stems from the fact that

1. All addresses are byte addresses.
2. Word addresses must be integer multiples of 4.

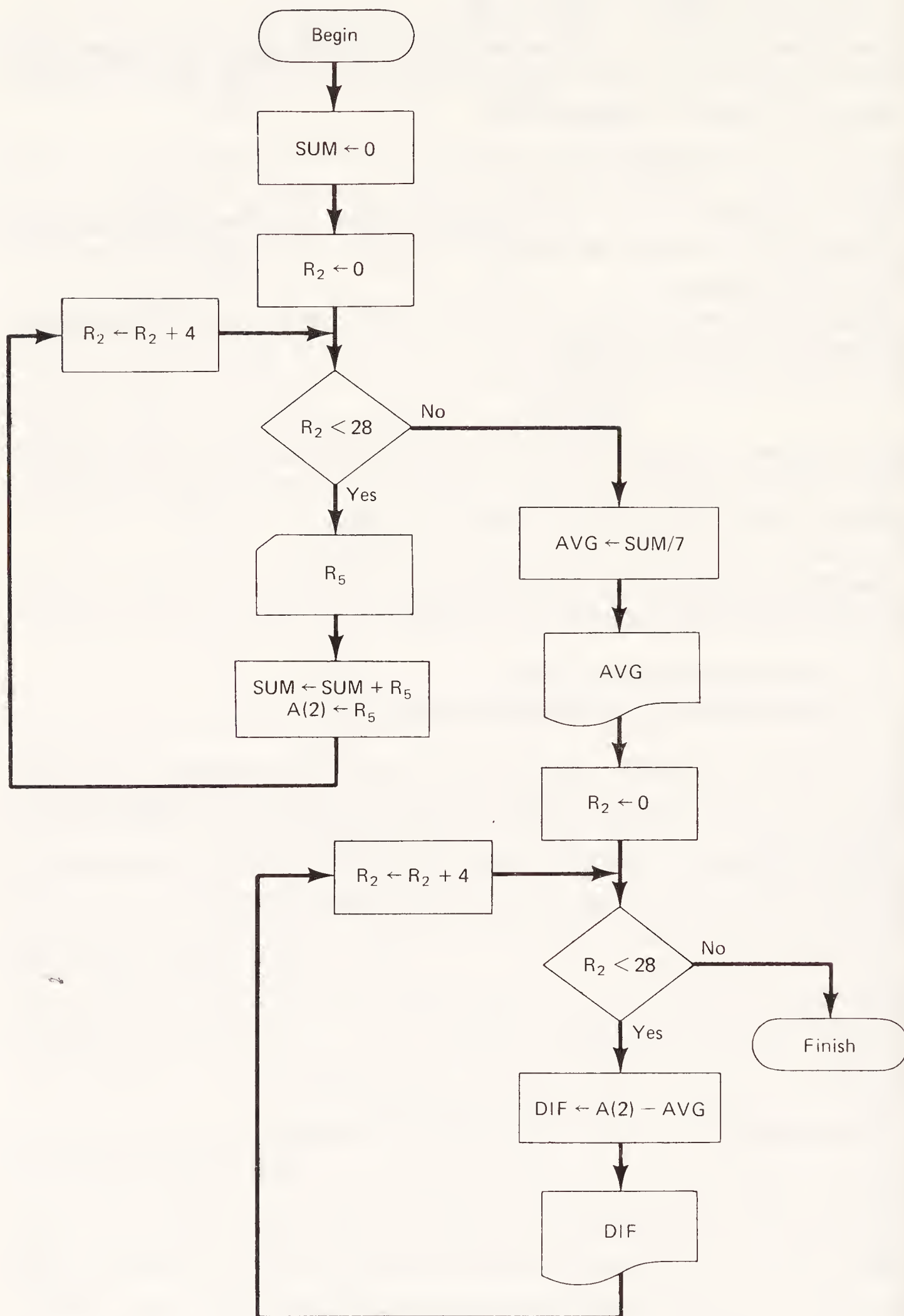
If the contents of register 3 in the above example contained, say, 2, an address mode error would result, since the address resulting from the indexing operation is not a multiple of 4 and therefore is not a valid word address.

In our program, the best place for the DS statement is between the EXIT and END statements, as will be illustrated in the following examples.

In Section 6.8, we illustrated the use of indexed addressing to compute the sum of seven numbers, compute their average, and then print the average followed by the difference of each of the numbers from the average. The flowchart and program are given in Figure A-9. In the flowchart, R_i designates the register i . In the ensuing discussion, the labels will be used to identify lines. That is, line READ+2 designates the statement BM AVG.

The statement in line 4 in the program loads into register 3 the value 4, which is used to increment the index register. Register 7 is used for variable SUM in the flowchart. The statement in line 5 initializes SUM to zero. Before entering the loop beginning with label READ, register 2, which will be used as the index register, is set to zero (line 6). To determine if it is time to exit from the loop, the value of the index register is subtracted from 28, and if the result is negative, a branch is made to AVG. The statements in lines READ+0, READ+1, and READ+2 serve this purpose.

In line READ+3, a number is read into register 5. In the following line, this value is stored in the areas reserved for A using 2 as an index register. The DS statement with label A reserves 7 words for A. In line READ+5, this value is



a. Flowchart

Figure A-9 Computation of average on the System 360

MAIN	START 0	
	PRINT NOGEN	
	INITIAL	
	L 3,=F'4'	REG 3 IS FOR CONSTANT 4
	L 7,=F'0'	REG 7 IS FOR SUM
	L 2,=F'0'	REG 2 IS FOR INDEXING
READ	L 4,=F'24'	
	SR 4,2	DECISION BLOCK FOR LOOP
	BM AVG	
	RWD 5	READ A NUMBER
	ST 5,A(2)	STORE
	AR 7,5	ADD NUMBER TO SUM
	AR 2,3	INCREMENT INDEX REGISTER
	B READ	
AVG	L 4,=F'7'	
	L 9,=F'1'	
	MR 8,7	EXTEND SUM TO DOUBLE WORD
	DR 8,4	DIVIDE
	WWD 9	WRITE AVERAGE
	L 2,=F'0'	SET INDEX REGISTER TO ZERO
WRITE	L 4,=F'24'	
	SR 4,2	DECISION BLOCK FOR LOOP
	BM OUT	
	L 5,A(2)	PUT NUMBER IN REGISTER
	SR 5,9	SUBTRACT AVERAGE
	WWD 5	WRITE DIFFERENCE
	AR 2,3	INCREMENT INDEX REGISTER
	B WRITE	
OUT	EXIT	
A	DS 7F	RESERVE SEVEN WORDS
	END MAIN	

b. Program

21
36
45
64
25
21
39

c. Input data

35
-14
1
10
29
-10
-14
4

d. Output

Figure A-9 (Continued)

432 added to SUM (register 6). The index register is then incremented (line READ+6), and a branch is made to the start of the loop (line READ+7).

Lines AVG+0 through AVG+3, divide SUM by 7. The number 7 is first loaded into register 4 (AVG+0), and the number 1 is loaded into register 9 (AVG+1). The MR instruction (AVG+2) effectively extends SUM (register 7) to a doubleword representation in registers 8 and 9. Dividing this result by the value 7 (in register 4) in line AVG+3 gives the average in Register 9. The result is written in line AVG+4.

The loop consisting of the eight statements beginning with label WRITE computes and prints the difference of each number from the average. Before entering this loop, the index register is reset to zero. Although we have chosen to use register 2 again, a different register could be used for indexing. The statements in lines WRITE+0, WRITE+1, and WRITE+2 determine when the exit is made from the loop. In line WRITE+3, a number is retrieved from memory and loaded into register 5. AVG (register 9) is then subtracted (line WRITE+4), and the result is printed (line WRITE+5). The index register is incremented (line WRITE+6), and a branch is made to the beginning of the loop (line WRITE+7).

A.13. INDEXED ADDRESSING
ON THE SIGMA 5/7 SERIES

As in the System 360, the basic unit of storage on the Sigma 5/7 Series is a byte. A word consists of four consecutive bytes with the restriction that the address of the first byte must be an integral multiple of 4. In this way, the last two bits of the address of the first byte in the word must be zero. Dropping these two bits produces the address of the word. In this way, word addresses contain two bits fewer than byte addresses. With the above restriction, however, both of these bits must be zero.

Although the Sigma 5/7 Series of computers recognize byte addresses, half-word addresses, word addresses, and doubleword addresses, for our purposes the computer can be considered as word-addressible. To reserve an area in memory, the appropriate statement is the RES statement, whose form is as follows for reserving words of memory:

```
label      RES,4      n
```

where *n* is the number of words reserved and label is associated with the address of the first word reserved. The 4 following RES designates reserving storage in units of 4 bytes or one word. In our programs, we can place the RES statements immediately after the REF statements.

Suppose we reserve a word of memory via the following statement:

```
VCT      RES,4      1
```

Information can be copied from one of the registers into this word of memory via the store-word instruction. For example, the instruction

STW,3 VCT

copies the information from register 3 into the word in memory reserved by the above statement. Similarly, the load-word instruction copies information from a word in memory into one of the registers. For example, the instruction

LW,5 VCT

copies the information in the word in memory reserved by the above RES statement into register 5.

On the Sigma 5/7 Series, registers 1 through 7 may be used as index registers. To indicate indexed addressing, the number of the index register follows the label of the area in memory, being separated by a comma. Indexed addressing may be used in either the LW or the STW instruction. For example, suppose we reserve 3 words in memory with the following statement:

VCT RES,4 3

In executing the statement

LW,6 VCT,2

the contents of register 2 would be added to the address associated with the label VCT to obtain the address of the word to be loaded into register 6. If register 2 contained zero, the first word in the reserved area would be loaded into register 6. If register 2 contained 1, the second word would be loaded into register 6. If register 2 contained 2, the third word would be loaded into register 6. Since word addressing is used, the index register must be incremented by 1 to proceed from one word in memory to the next.

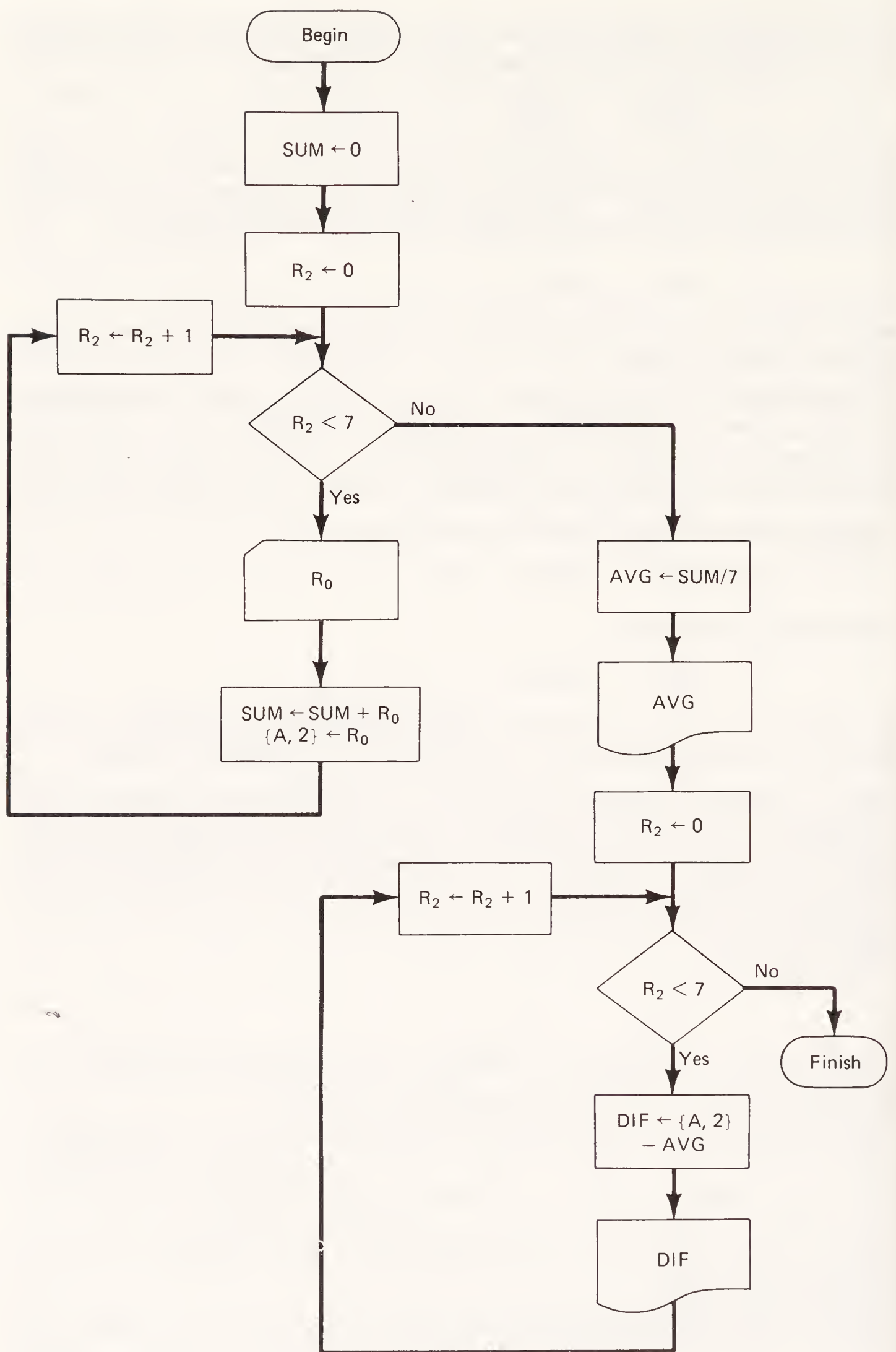
The instruction

STW,6 VCT,2

is similar to the LW instruction described in the preceding paragraph except that information is copied from the register into memory.

To illustrate indexed addressing on the Sigma series, we shall prepare a program to read seven numbers, compute their average, and print the difference of each of the original numbers from the average. The flowchart and program are given in Figure A-10. In the flowchart, R_i designates register i . In the ensuing discussion the labels will again be used to identify lines. That is, READ+2 designates the statement BL AVG.

In line 3, the RES statement reserves seven words for A . The constant 1 (to be used to increment the index register) is loaded into register 2 by the statement with label MAIN. In line MAIN+1, register 7 (to be used for SUM) is initialized to zero. Before entering the loop beginning with line READ+0, the index register, namely, register 2, is initialized to zero.



a. Flowchart

Figure A-10 Computation of average on the Sigma 5/7 series

	REF	RWD	
	REF	WWD	
A	RES,4	7	RESERVE SEVEN WORDS
MAIN	LI,3	1	REG 3 IS FOR CONSTANT 1
	LI,7	0	REG 7 IS FOR SUM
	LI,2	0	REG 2 IS FOR INDEXING
READ	LI,4	6	
	SW,4	2	DECISION BLOCK FOR LOOP
	BL	AVG	
	BAL,15	RWD	READ A NUMBER
	STW,0	A,2	STORE
	AW,7	0	ADD NUMBER TO SUM
	AW,2	3	INCREMENT INDEX REGISTER
	B	READ	
AVG	LI,4	7	
	DW,7	4	COMPUTE AVG
	STW,0	7	
	BAL,15	WWD	WRITE
	LI,2	0	SET INDEX REGISTER TO ZERO
WRITE	LI,4	6	
	SW,4	2	DECISION BLOCK FOR LOOP
	BL	OUT	
	LW,0	A,2	PUT NUMBER IN REGISTER
	SW,0	7	SUBTRACT AVERAGE
	BAL,15	WWD	WRITE
	AW,2	3	INCREMENT INDEX REGISTER
	B	WRITE	
OUT	CAL1,9	1	
	END	MAIN	

21
36
45
64
25
21
39

b. Input data

39
-18
-3
6
25
-14
-18
0

c. Program and output

Figure A-10 (Continued)

Lines READ+0, READ+1, and READ+2 constitute the decision block for the loop. The constant 7 is loaded into register 4, the current value of the index register is then subtracted, and if the result is negative, an exit is made from the loop. In line READ+3, a number is read into register 0, is stored in A in line READ+4, and added to SUM (register 7) in line READ+5. The index register is incremented (line READ+6) and a branch made (line READ+7) to the start of the loop.

Lines AVG+0 and AVG+1 compute the average by first loading the constant 7 into register 4 and then dividing into SUM. Since SUM is in an odd-numbered register, the quotient is obtained directly in register 7. The result is moved to register 0 (line AVG+2) and written (line AVG+3).

Before entering the loop beginning with line WRITE+0, the index register is reset to zero in line AVG+4. The statements in lines WRITE+0, WRITE+1, and WRITE+2 constitute the decision block for this loop. In line WRITE+3, a number is loaded into register 0, AVG (register 7) is subtracted in line WRITE+4, and the result written in line WRITE+5. The index register is incremented (line WRITE+6), and a branch is made (line WRITE +7) to the top of the loop.

**BASIC**

Developed with ease of learning as primary objective, Basic provides an avenue by which a beginning student with no previous computer experience can very quickly begin to write programs and use the computer. Although a few systems offer Basic in the batch-processing mode, Basic is more commonly available on time-sharing systems; therefore, the discussion in this chapter will assume that time-sharing is the *modus operandi*.

In the introduction to Basic in this appendix we shall follow the same order as was followed in Chapters 2 through 5 of this text. In general, the objective is to introduce features in the sequence that will permit the student to begin writing programs as early as possible.

B.1. CONSTANTS AND VARIABLES

In a Basic program, a constant is a string of digits that may be written with or without a decimal point and may or may not be preceded by a sign (unsigned constants are considered positive). Valid constants include the following:

12.45
127
-147.
+4472
.000012765
1476530000.

Most systems permit constants to contain up to seven significant digits (that is, not counting leading or trailing zeros). The constant .000012765 contains five significant digits; the constant 1476530000 contains six significant digits.

For especially large or small constants, the scientific or exponential representation of the constant is often more convenient. For example, the number

438 1,476,530,000 can be expressed as $147,653 \times 10^4$. In Basic, this representation is denoted by the number followed by the letter E followed by the exponent (which must be an integer number written without a decimal point). The following examples are valid:

$147,653 \times 10^4$	147653E4
1.47653×10^9	1.47653E + 9
1.2765×10^{-5}	1.2765E -5
127.65×10^{-7}	127.65E -7

On most systems, the exponent must be between +50 and -50. The letter E must be preceded by a digit. As we shall see in the next paragraph, E2 is a variable, not the constant 100 (which could be represented as 1E2).
Variables in Basic are represented as a letter of the alphabet, optionally followed by one of the ten digits. Valid examples include

A
G
E2
B5
C0
Z9

Invalid examples include

SUM	(too many alphabetic characters)
A14	(too many digits)
1X	(first character is not an alphabetic character)

This provides a maximum of $26 \times 11 = 286$ variables, which is sufficient for all but the very largest problems.

B.2. ARITHMETIC OPERATIONS

Arithmetic operations are designated by the following symbols:

Symbol	Operation	Example	Explanation
+	Addition	$A + B$	<i>A</i> plus <i>B</i>
-	Subtraction	$A - B$	<i>A</i> minus <i>B</i>
-	Negation	-A	minus <i>A</i>
*	Multiplication	$A * B$	<i>A</i> times <i>B</i>
/	Division	A / B	<i>A</i> divided by <i>B</i>
↑ or **	Exponentiation	$A \uparrow B$ or $A ** B$	<i>A</i> raised to the <i>B</i> power

The hierarchy of operations is as described in Section 2-7.

In performing exponentiations, negative values can be raised only to integer powers. For nonintegral powers, exponentiation is performed as follows:

$$A^B = 10^{\uparrow(B \cdot \log_{10} A)}$$

For real numbers, the function $\log_{10} A$ exists only for positive A .

Examples of valid Basic expressions are the following:

Basic expression	Algebraic interpretation
C * 2 + B	$c * 2 + b$
C + B/2	$c + b/2$
C/2 + B	$c/2 + b$
C ↑ 2/D	c^2/d
-C ↑ 2	$-(c^2)$, not $(-c)^2$

B.3. INTRINSIC FUNCTIONS

Table B-1 gives a list of the intrinsic or library functions commonly available in Basic. All functions have three-letter names followed by an argument enclosed in

Table B-1 Common Basic Functions

Function	Explanation
SIN (X)	Calculates trigonometric sine of X , where X is in radians
COS (X)	Calculates trigonometric cosine of X , where X is in radians
TAN (X)	Calculates trigonometric tangent of X , where X is in radians
ATN (X)	Calculates trigonometric arctangent of X . Result is in radians, and is positioned as follows: $X < 0 - \frac{\pi}{2} < \text{result} < 0$ $X \geq 0 \quad 0 \leq \text{result} < \frac{\pi}{2}$
EXP (X)	Calculates the exponential function, e^x
LOG (X)	Calculates natural logarithm (base e) of X
LGT (X)	Calculates common logarithm (base 10) of X
ABS (X)	Calculates absolute value of X , $ X $
SQR (X)	Calculates square root of X , $\sqrt{X}$
INT (X)	Calculates the integer part of X (See text for explanation.)
SGN (X)	Calculates the sign of X – that is, if $X < 0$ SGN (X) = -1 if $X = 0$ SGN (X) = 0 if $X > 0$ SGN (X) = +1

parentheses. The argument may be a constant, a variable, or any valid expression, as illustrated by the following examples:

```
SQR (4*Y + 9)
INT (A/B)
LOG (10) + Q
X + EXP(Y)
ABS (SIN (4*G))
SQR(SQR(H))
```

The last examples illustrate that functions may appear in an expression used as the argument of the same function or another function.

The use of the INT function varies from system to system, as noted in Section 2.5. On most Basic systems, the INT function “determines the largest integer number that does not exceed X.” In a few systems, however, the INT function truncates fractional parts. In the Basic programs in this Appendix, we shall take the former interpretation.

B.4. THE LET STATEMENT

The arithmetic assignment statement in Basic takes the following form:

LET variable = expression

This statement causes the expression on the right of the equals sign to be calculated and the result assigned to the variable on the left of the equals sign (which serves as the replacement operator in Basic). Only variables may appear on the left of the equals sign. Valid examples are the following:

```
LET Y = G
LET Z2 = SQR(X4) - 10
LET A9 = A9 - 17
```

B.5. THE INPUT STATEMENT

For entry of data from the terminal into a Basic program during the execution phase, the appropriate statement is the INPUT statement:

INPUT variable list

where *variable list* is a list of variables separated by commas. For example, values for variables A, X2, and J can be entered using the following statement

```
INPUT A, X2, J
```

Note that no comma follows the word INPUT.

When the INPUT statement is executed, the ? character is typed by the computer, which then waits for the programmer to enter numerical values for the respective variables. Any representation acceptable as a constant can be used, and the values should be separated by commas. For example, entry of data in response to the statement

```
INPUT  A, X2, J
```

might be as follows:

```
15, 1.2E4, -.027
```

The value 15 is assigned to A, 1.2E4 to X2, and -0.027 to J.

B.6. THE READ AND DATA STATEMENTS

The READ statement is identical to the INPUT statement except that the values of the constants to be assigned to the respective variables are obtained from the DATA statement instead of being entered from the teletype. The formats of these two statements are as follows:

```
READ variable, variable, etc.
DATA constant, constant, etc.
```

Neither of these statements should appear in a Basic program without at least one occurrence of the other.

Although a program may contain several DATA statements, they are essentially considered to comprise a single block of data for the READ statement(s). That is, the DATA statements

```
DATA  1,5.7
DATA  -2.4E - 7,8.1
DATA  5.2
```

are equivalent to the single DATA statement

```
DATA  1,5.7, -2.4E - 7,8.1,5.2
```

The Basic system essentially maintains a “pointer” to the next available constant. Initially, this pointer is set to the first constant in the first DATA statement.

For example, suppose the DATA statements are as given above, and that the first READ statement is as follows:

```
READ  A,X
```

Variable A is assigned the value 1, variable X is assigned the value 5.7, and the pointer is moved to the constant -2.4E-7. If this same READ statement is executed again, variable A is assigned the value -2.4E-7, variable X is assigned

the value 8.1, and the “pointer” is moved to the constant 5.2. If this same READ statement is executed a third time, the constant 5.2 is available for variable A, but no value is available for variable X. This generates an OUT OF DATA message, after which execution is terminated.

Should the data for a particular problem need to be entered a second time, a statement is available to reset the pointer to the first constant in the first DATA statement. This is the RESTORE statement, which is an executable statement that would occupy a line of the program.

The DATA statements are nonexecutable statements, and their location within the program is immaterial. The result is as if the BASIC system collected all of the constants in the DATA statements into a single block of data to which all READ statements refer. Many programmers prefer to place all of their DATA statements near the end of the program, but this is not necessary.

B.7. THE PRINT STATEMENT

The primary output statement in Basic is the PRINT statement, whose format parallels that of the INPUT and READ statements:

```
PRINT variable, variable, etc.
```

The PRINT statement may, however, contain either constants or expressions in place of one or more of the variables. The following examples are valid:

PRINT 2	(prints the constant 2)
PRINT A, B*C	(prints the value of variable A and the result of the evaluation of the expression B*C)

Only numerical values are printed; that is, the variable names or other descriptive material are not automatically printed.

In the PRINT statement, the commas play a greater role than just separating the variables. In effect, the output line is divided into five fields, each of which is fifteen columns wide. The appearance of a comma advances the output typer from one field to the next. For example, the statement

```
PRINT A
```

prints the numerical value of variable A in the first field. The statement

```
PRINT A,, B
```

prints the value in the second field. The statement

```
PRINT A, B
```

prints the value of A in the first field, nothing in the second field, and the value of B in the third field.

If a PRINT statement calls for more than five fields, a carriage return occurs after the fifth field. For example, the statement

PRINT,A,,B,C

prints the value of A in the second field of the first line, variable B in the fifth field of the first line, and variable C in the first field of the second line.

Should a comma appear after the last variable (or constant or expression) in a PRINT statement, the carriage return does not occur. For example, the statement

PRINT A

prints the value of A in the first field and returns the carriage so that the first field of the next line is available for the next output. On the other hand, the statement

PRINT A,

prints the value of variable A in the first field and does *not* return the carriage. Therefore, the second field of that line is available for the next output.

The statement

PRINT

causes the carriage to be returned. If the last PRINT statement executed prior to this one returned the carriage, a blank line is obtained. The statement

PRINT,

advances the typer one field.

Messages can be printed by simply enclosing the desired characters in quotation marks. For example, the statement

PRINT "HERE WE GO"

prints the message HERE WE GO. Messages can be interspersed with variables, constants, or expressions. For example, the PRINT statement

PRINT "A =" A

prints the message A = followed immediately by the numerical value of variable A. Note that no comma appears between the message and the variable in the PRINT statement. If a comma is inserted as in the statement

PRINT "A =",A

the message A = is printed in the first field and the value of variable A is printed in the second field. In effect, the appearance of the comma advances the typer to the next field, thereby permitting messages longer than one field in width to be accommodated.

Column headings can be very readily obtained in Basic. For example, the statements

PRINT "R1", "R2", "R3"

PRINT R1, R2, R3

would provide the following output

R1	R2	R3
1.276	-1.41	91.722

The combination of messages and commas generally provides enough flexibility to obtain the desired format for the output.

In printing numerical values, Basic normally prints six digits. If the number is between 0.1 and 999999., it is written in the customary decimal format with trailing zeros deleted from fractions. Numbers outside this range are written in the exponential form with the exception that numbers between 0.00001 and 0.1 will be written in decimal format provided that no significant digits are lost. That is, the number 0.001500 would be written as 0.0015, whereas the number 0.0015001 would be written as 0.15001 E-2.

The use of semicolons in place of commas in PRINT statements instructs the computer to use packed format for output. This causes numerical values to be written much closer together than in the normal format, thereby providing more output per line.

B.8. THE REM STATEMENT

The programmer may insert explanatory information into his program by use of the REM statement, whose format is as follows:

```
REM  comments
```

A valid example is the following:

```
REM  WE START HERE
```

This statement is not executable and has absolutely no effect upon the execution of the program. REM statements can be inserted at any point in a program.

B.9. THE END STATEMENT

The very last statement in every Basic program is the END statement. Execution is terminated upon processing the END statement.

B.10. A BASIC PROGRAM

As discussed in Section 1-5, every line in a Basic program must begin with a line number. The line number consists of from one to five digits with no imbedded blanks. The line number must be followed by at least one blank space, and thereafter blanks have no effect on statements in Basic (except when they appear in messages in PRINT statements). For example, the statement

150 L ETR 1 = A 2*B

is equivalent to

150 LET R1 = A2*B

Of course, the last example is much more readable.

Basic always executes statements in ascending order of their line numbers, regardless of the order in which the statements were entered. As illustrated by the example in Figure 1-3, this permits statements to be readily added or deleted. Entering a statement with the same line number as one already in the program causes the system to replace the old statement with the new one. When the system is directed to list the program, the statements are listed in ascending order of the line numbers.

In specifying line numbers for statements in Basic, an increment of 10 is generally used for the line numbers in the original program. In this way, statements can be readily added without renumbering the lines in the entire program. Actually, even this is quite simple with the editing features available at the terminal.

As our first example of a Basic program, we shall use the problem in Exercise 2-22, which entails the computation of the minimum number of coins in change for purchases under \$1.00. Only pennies, nickels, dimes, and quarters are used, and the purchaser always submits a dollar bill. Figure B-1a presents the solution of this problem using the INPUT statement, and Figure B-1b presents the solution using the READ statement.

```
LISTNH
10 INPUT B
20 LET P=100-B
30 LET Q=INT(P/25)
40 LET P=P-25*Q
50 PRINT "CHANGE IS"
60 PRINT Q,"QUARTERS"
70 LET D=INT(P/10)
80 LET P=P-10*D
90 PRINT D,"DIMES"
100 LET N=INT(P/5)
110 LET P=P-5*N
120 PRINT N,"NICKELS"
130 PRINT P,"PENNIES"
140 END
```

```
READY
RUNNH
? 39
CHANGE IS
2          QUARTERS
1          DIMES
0          NICKELS
1          PENNIES
```

a. Use of INPUT statement

```
LISTNH
10 READ B
20 LET P=100-B
30 LET Q=INT(P/25)
40 LET P=P-25*Q
50 PRINT "CHANGE IS"
60 PRINT Q,"QUARTERS"
70 LET D=INT(P/10)
80 LET P=P-10*D
90 PRINT D,"DIMES"
100 LET N=INT(P/5)
110 LET P=P-5*N
120 PRINT N,"NICKELS"
130 PRINT P,"PENNIES"
140 DATA 39
150 END
```

```
READY
RUNNH
CHANGE IS
2          QUARTERS
1          DIMES
0          NICKELS
1          PENNIES
```

b. Use of READ-DATA statements

Figure B-1 Basic program to compute coins in change

In each run, the first step is to enter the complete Basic program, which is then debugged as illustrated in Figure 1-19. For all the example programs in this section, we shall first list the debugged program and then run the program to show the output produced. Figure B-1 presents two such programs. Note that each statement (both executable and non-executable) in the Basic program is given a line number and that we have chosen an increment of 10 for the line numbers.

B.11. TRANSFER OF CONTROL

Basic offers three statements that effect a transfer of control:

1. Unconditional GOTO
2. The IF-THEN statement
3. The ON-GOTO statement.

Each of these statements causes the computer to execute some statement other than the one with the next highest line number.

Unconditional GOTO

The format of this statement is

GOTO *n*

where *n* is the line number of another statement in the Basic program. Variables are not allowed in place of *n*.

The IF-THEN Statement

The format of this statement is as follows:

IF *relational expression* THEN *n*

where *n* is the line number of another statement in the Basic program and *relational expression* is a statement whose value is either TRUE or FALSE. If the *relational expression* is TRUE, the statement whose line number is *n* is executed next. If *relational expression* is FALSE, the statement with the next highest line number is executed.

The formulation of the relational expression is analogous to that indicated in Section 2-4. Table B-2 summarizes the formulation of relational expressions in Basic. Compound conditionals such as described in Section 2-8 are not permitted in Basic.

To illustrate the use of the IF-THEN statement and the unconditional GOTO statement in Basic, Figure B-2 gives the program for the flowchart in Figure 2-12 for ascertaining if a number is prime. This program also makes use of the STOP statement in Basic, which essentially terminates the execution of the program in

Table B-2 Relational Symbols

Symbol	Explanation	Example
=	Equal to	IF X = Y THEN 30; Control is transferred to line 30 only if X equals Y.
<> or <>	Not equal to	IF X <> Y THEN 30 or IF X >< Y THEN 30; Control is transferred to line 30 only if X is not equal to Y.
<	Less than	IF X < Y THEN 30; Control is transferred to line 30 only if X is less than Y.
>	Greater than	IF X > Y THEN 30; Control is transferred to line 30 only if X is greater than Y.
<= or =<	Less than or equal	IF X <= Y THEN 30 or IF X = < Y THEN 30; Control is transferred to line 30 only if Y is greater than X.
>= or =>	Greater than or equal	IF X >= Y THEN 30 or IF X = > Y THEN 30; Control is transferred to line 30 only if Y is less than X.

the same fashion as the END statement. In Basic, the STOP statement is equivalent to an unconditional GOTO statement transferring control to the END statement.

The ON-GOTO Statement

Being Basic’s counterpart to multiple branching as discussed in Section 3.6, the ON-GOTO statement has the following format:

ON expression GOTO $n_1\ n_2, n_3, \dots, n_k$

```
LISTNH
10 INPUT M
20 LET X=M/2
30 LET N=2
40 LET F=M/N
50 IF F=INT(F) THEN 100
60 LET N=N+1
70 IF N<=X THEN 40
80 PRINT M,"IS A PRIME NUMBER"
90 STØP
100 PRINT M,"IS NØT A PRIME NUMBER"
110 END

READY
RUNNH
? 313791
313791          IS NØT A PRIME NUMBER
```

Figure B-2 Prime number program in Basic

448 Processing of this statement proceeds as follows:

1. The expression is evaluated
2. Fractional parts are deleted
3. If the result is 1, the next statement to be executed is the one with line number n_1 , if the result is 2, the next statement to be executed is the one with line number n_2 ; etc.

If the result of the expression (with fractional parts truncated) is less than 1 or greater than k , an error message is generated.

Figure B-3 gives the Basic program corresponding to the flowchart in Figure 3-16b. This program uses a zero entry to indicate end of data.

```

LISTNH
10 LET T=0
20 LET F=0
30 LET S0=0
40 LET J=0
50 LET S=0
60 READ A,C
70 IF A=0 THEN 180
80 LET T=T+A
90 ON C GOTO 100,120,140,160
100 LET F=F+1
110 GOTO 60
120 LET S0=S0+1
130 GOTO 60
140 LET J=J+1
150 GOTO 60
160 LET S=S+1
170 GOTO 60
180 LET D=F+S0+J+S
190 LET A1=T/D
200 PRINT "AVERAGE AGE IS",A1
210 PRINT F*100/D;"% FRESHMEN"
220 PRINT S0*100/D;"% SOPHOMORES"
230 PRINT J*100/D;"% JUNIORS"
240 PRINT S*100/D;"% SENIORS"
250 DATA 23,2,19,1,25,3,22,4,18,1,29,3,18,1
260 DATA 20,2,20,1,25,4,22,3,0,0
270 END

```

```

READY
RUNNH
AVERAGE AGE IS  21.9091
 36.3636 % FRESHMEN
 18.1818 % SOPHOMORES
 27.2727 % JUNIORS
 18.1818 % SENIORS

```

Figure B-3 Example of the use of the ON-GOTO statement

B.12. THE FOR-NEXT STATEMENTS

In BASIC the counterpart to the iteration counter described in Chapter 3 is the FOR-NEXT pair of statements whose formats are as follows:

```
FOR index =  $expr_1$  TO  $expr_2$  STEP  $expr_3$ 
NEXT index
```

where *index* is the variable to be used as the index of the loop, *expr₁* designates the initial value of *index* on the first pass around the loop; *expr₂* designates the final value of *index*, and *expr₃* designates the amount by which *index* is to be incremented after each pass through the loop. Control is as per Version II of the iteration counter in Figure 3-7.

The FOR and NEXT statements must always appear as a pair of statements with the statements that comprise the loop placed in between.

As our first example of the use of the FOR-NEXT statements, the program in Figure B-4 corresponds to the flowchart in Figure 3-9 for computing an entry in the Fibonacci sequence. The loop begins with the FOR statement in line 40 and ends with the NEXT statement in line 80. After executing the loop the desired number of times, the next statement to be executed is the one following the NEXT statement.

In the FOR statement, omission of the STEP *expr₃* portion implies an increment of one. For example, the FOR statement in line 40 of Figure B-4 can be shortened to

```
FOR J = 3 TO N
```

As our next example of the use of the FOR-NEXT pair of statements, Figure B-5 presents the Basic program corresponding to the flowchart in Figure 3-11. This program illustrates that in Basic the value of a variable appearing in one of

```
LISTNH
10 LET E1=1
20 LET E2=0
30 INPUT N
40 FOR J=3 TO N STEP 1
50 LET E=E1+E2
60 LET E2=E1
70 LET E1=E
80 NEXT J
90 PRINT "ENTRY ";N;" IS ";E
100 END
```

```
READY
RUNNH
? 19
ENTRY 19 IS 2584
```

Figure B-4 Fibonacci program in Basic

450

```

LISTNH
10 INPUT N
20 PRINT "FACTORS OF ",N
30 FOR J=2 TO N
40 LET K=INT(N/J)
50 IF J*K<>N THEN 90
60 PRINT J;" IS A FACTOR"
70 LET N=K
80 GOTO 40
90 NEXT J
100 END

```

```

READY
RUNNH
? 45
FACTORS OF      45
3  IS A FACTOR
3  IS A FACTOR
5  IS A FACTOR

```

Figure B-5 Example of the use of the FOR-NEXT statement

the expressions used as an indexing parameter in the FOR statement may be changed by the statements within the loop. That is, the variable N appears as an indexing parameter in the FOR statement in line 10 but is assigned new values by statement 60 which is inside the loop. In general, Basic is very flexible in regard to the indexing parameters, permitting the index to assume positive or negative values, permitting the step to be a nonintegral value, etc.

The Basic program in Figure B-6 corresponds to the flowchart in Figure 3-13 and illustrates a nested pair of FOR-NEXT statements. As discussed in Section 3.4, one of the loops must lie completely within the other; that is, they must not overlap. Furthermore, the index of the inner loop must not be the same as the index for the outer loop.

B.13. SUBSCRIPTED VARIABLES

In Basic, subscripts are enclosed in parentheses and follow the variable name, which is restricted to being a single alphabetic character (giving a total of twenty-six possibilities). Subscripts may be constants, variables, or expressions. Should the value of the subscript be nonintegral, the fractional part is truncated. Valid examples of the use of subscripted variables are as follows:

```

A(4)
B(G)
X(4*M - D)

```

Basic also permits statements such as the following:

```
LET A = A + A(4)
```

where A written without a subscript denotes a simple variable and A written with a subscript designates an array.

```

LISTNH
10 FOR A=5780 TO 5790 STEP 2
15 LET N=A
20 PRINT "FACTORS OF ",N
30 FOR J=2 TO N
40 LET K=INT(N/J)
50 IF J*K<>N THEN 90
60 PRINT J;" IS A FACTOR"
70 LET N=K
80 GOTO 40
90 NEXT J
95 NEXT A
100 END

```

```

READY
RUNNH
FACTORS OF      5780
 2  IS A FACTOR
 2  IS A FACTOR
 5  IS A FACTOR
17  IS A FACTOR
17  IS A FACTOR
FACTORS OF      5782
 2  IS A FACTOR
 7  IS A FACTOR
 7  IS A FACTOR
59  IS A FACTOR
FACTORS OF      5784
 2  IS A FACTOR
 2  IS A FACTOR
 2  IS A FACTOR
 3  IS A FACTOR
241 IS A FACTOR
FACTORS OF      5786
 2  IS A FACTOR
11  IS A FACTOR
263 IS A FACTOR
FACTORS OF      5788
 2  IS A FACTOR
 2  IS A FACTOR
1447 IS A FACTOR
FACTORS OF      5790
 2  IS A FACTOR
 3  IS A FACTOR
 5  IS A FACTOR
193 IS A FACTOR

```

Figure B-6 Example of the use of nested FOR-NEXT statements

Figure B-7 gives the BASIC program corresponding to the flowchart in Figure 4-2. The first loop reads the values into array A and sums them. The second loop computes and prints the differences from the average. Note the use of A as the name of a subscripted variable and as the name of a simple variable. Also note the use of an expression in the PRINT statement in line 100.

Basic automatically reserves ten storage locations for each array. In the program in Figure B-7, we used only seven of them. Should one wish to reserve

452

```
LISTNH
10 LET S=0
20 FOR J=1 TO 7
30 READ A(J)
40 LET S=S+A(J)
50 NEXT J
60 LET A=S/7
70 PRINT "AVERAGE AGE IS",A
80 PRINT "STUDENT","AGE","DIFFERENCE"
90 FOR J=1 TO 7
100 PRINT J,A(J),A(J)-A
110 NEXT J
120 DATA 21,36,45,64,25,21,39
130 END

READY
RUNNH
AVERAGE AGE IS 35.8571
STUDENT      AGE      DIFFERENCE
1            21      -14.8571
2            36       .142857
3            45       9.14286
4            64      28.1429
5            25     -10.8571
6            21     -14.8571
7            39       3.14286
```

Figure B-7 Use of subscripted variables in Basic

more or fewer storage locations, the DIM statement can be used. For example, the statement

```
DIM A(7)
```

could be used with the program in Figure B-7 to cause only seven locations to be reserved for array A. Most programmers tend to place the DIM statement at the very beginning of the program. The size of several arrays can be specified in the same DIM statement (as will be illustrated in the next example), and more than one DIM statement may be used if desired.

As our next example of the use of subscripted variables, the program in Figure B-8 corresponds to the flowchart in Figure 4-3. The DIM statement in line 10 reserves 20 locations for arrays S and Q. Basic does not have a special input or output statement for arrays such as illustrated in Section 4.5; therefore, the READ and PRINT statements are placed inside a loop. Zero entries are used to indicate “end-of-data” for the program in Figure B-8.

Many versions of Basic reserve a storage location for the zero subscript but never for negative subscripts. Before writing a program using zero subscripts, a programmer should first be sure that the system he will use reserves a storage location for the zero subscript.

Basic permits an array to have a maximum of two subscripts. For double-subscripted arrays, storage is reserved for a 10 X 10 array, or a total of 100 storage locations for systems that do not recognize the zero subscript. On

```

LISTNH
10 DIM S(20),Q(20)
20 READ I
30 FOR J=1 TO I
40 READ S(J),Q(J)
50 NEXT J
60 READ N,R
70 IF N=0 THEN 150
80 FOR J=1 TO I
90 IF N=S(J) THEN 130
100 NEXT J
110 PRINT "NO ITEM NUMBER";N
120 GOTO 60
130 LET Q(J)=Q(J)+R
140 GOTO 60
150 PRINT "ITEM","QUANTITY"
160 FOR J=1 TO I
170 PRINT S(J),Q(J)
180 NEXT J
190 DATA 8,1102,19,1543,21,1011,5,1917,8
200 DATA 1414,10,1998,3,2122,14,1201,8
210 DATA 1998,6,2122,5,1011,7,1534,9,1917,5
220 DATA 1102,8,1543,19,1998,3,1201,3,1102,5,0,0
230 END

```

READY
RUNNH

NO ITEM NUMBER 1534	
ITEM	QUANTITY
1102	32
1543	40
1011	12
1917	13
1414	10
1998	12
2122	19
1201	11

Figure B-8 The inventory program in Basic

systems that recognize the zero subscript, a total of 121 (11 X 11) locations are reserved automatically. As for single-subscripted arrays, the DIM statement can be used to instruct the Basic system to reserve more or fewer storage locations. The use of double-subscripted arrays in BASIC is quite analogous to the use of single-subscripted arrays.

B.14. CHARACTER DATA

Variables used to represent character data are designated by a letter of the alphabet followed by the \$ character. For example, the variable X\$ would be used to store character data. In each variable, a maximum of around fifteen or

eighteen characters (depending upon the system) may be stored. For character data, LET statements of the type

```
LET A$ = "STICK EM UP"
LET B$ = A$
```

are valid, but character variables may not appear in arithmetic expressions. The relational operators of Table B-2 are also valid for character data, the "greater than" and "less than" decisions depending upon the collating sequence in effect on the particular system. In this appendix, we shall only be interested in the "is equal to" and "is not equal to" decisions, which apply uniformly to all systems.

Most systems permit variables for character data to be single-subscripted but not double-subscripted.

As our first example of the use of character variables, we shall write a program to convert from temperature in degrees Fahrenheit to temperature in degrees centigrade and vice versa. As input, a numerical value for the temperature will be entered followed by an F or a C to indicate degrees Fahrenheit or centigrade. The program is given in Figure B-9. The temperature and the letter designating Fahrenheit or centigrade are input in line 10. The variable T\$ is compared with an "F" in line 10 and a "C" in line 30, branching to the appropriate section of the program upon equality. If the letter is neither an F nor a C, an "error message" is printed in line 40.

To illustrate that Basic is capable of performing operations on nonnumeric data, the Basic program corresponding to the flowchart in Figure 4-17 for cryptogram enciphering is given in Figure B-10. In writing this program, we have

```
LISTNH
10 INPUT T,T$
20 IF T$="F" THEN 60
30 IF T$="C" THEN 100
40 PRINT "INVALID INPUT",T$
50 GOTO 10
60 LET F=T
70 LET C=(F-32)/1.8
80 PRINT "TEMPERATURE IS ";F;" DEG F OR ";C;" DEG C"
90 GOTO 10
100 LET C=T
110 LET F=1.8*C+32
120 GOTO 80
130 END
```

```
READY
RUNNH
? 150,"F"
TEMPERATURE IS 150 DEG F OR 65.5556 DEG C
? 38.5,"D"
INVALID INPUT D
? 38.5,"C"
TEMPERATURE IS 101.3 DEG F OR 38.5 DEG C
? S
```

Figure B-9 Example of the processing of character data in Basic

assumed that our system reserves a storage location for the zero subscript. On systems that do not reserve this storage location, the flowchart in Figure 4-17 must be modified slightly.

```

LISTNH
10 DIM S$(26),K$(8),M$(16),C$(16),N(8)
20 FOR J=0 TO 26
30 READ S$(J)
40 NEXT J
50 FOR J=1 TO 8
60 READ K$(J)
70 NEXT J
80 FOR J=1 TO 16
90 READ M$(J)
100 NEXT J
110 FOR J=1 TO 8
120 FOR N=0 TO 26
130 IF K$(J)=S$(N) THEN 170
140 NEXT N
150 PRINT "INVALID CHARACTER IN KEYWORD",K$(J)
160 STOP
170 LET N(J)=N
180 NEXT J
190 LET J=0
200 FOR L=1 TO 16
210 LET J=J+1
220 IF J<>9 THEN 240
230 LET J=1
240 FOR N=0 TO 26
250 IF M$(L)=S$(N) THEN 290
260 NEXT N
270 PRINT "INVALID CHARACTER IN MESSAGE",M$(L)
280 STOP
290 LET M=N+N(J)+10
300 LET M=M-27*INT(M/27)
310 LET C$(L)=S$(M)
320 NEXT L
330 PRINT "CRYPTOGRAM IS",
340 FOR L=1 TO 16
350 PRINT C$(L);
360 NEXT L
400 REM DATA FOR ARRAY S$
410 DATA " ","A","B","C","D","E","F","G","H","I","J"
420 DATA "K","L","M","N","O","P","Q","R","S","T","U"
430 DATA "V","W","X","Y","Z"
440 REM KEYWORD
450 DATA "C","O","M","P","U","T","E","R"
460 REM MESSAGE
470 DATA "H","E"," " ,"H","A","D"," " ,"A"," "
480 DATA "B","A","D"," " ,"D","A","Y"
490 END

```

```

READY
RUNNH
CRYPTOGRAM IS  UCWGEGBM XCDGPZ

```

Figure B-10 The enciphering program in Basic

B.15. FUNCTIONS

Basic permits a programmer to define up to twenty-six functions. The name of all such user-defined functions consists of the letters FN followed by a single letter of the alphabet. A special statement known as the DEF statement is used to define functions.

As an example of a user-defined function, suppose you would like a function to convert a temperature in degrees centigrade to a temperature in degrees Fahrenheit, where

$$^{\circ}\text{F} = 1.8 \times ^{\circ}\text{C} + 32$$

A function would be defined in Basic as follows:

```
DEF FNF(T) = 1.8*T + 32
```

This statement only defines the function, and therefore is a nonexecutable statement. The function is executed whenever the statement in which the function FNF appears is executed, such as the following:

```
LET D = 212 - FNF(P)
```

Whenever this statement is executed, the function is executed.

Although most Basic systems accept DEF statements anywhere, most programmers tend to place them near the beginning. Some Basic systems permit user-defined functions to have more than one argument, but in many systems only one argument is allowed.

The flowchart in Figure 4-11b is a good example where a user defined function can be advantageous. In the Basic program in Figure B-11, the function is defined by the DEF statement in line 10, and then called from the PRINT statements in lines 40, 60, and 80. Since this problem calls for six columns of output and only five fields are generally available on each output line, we have introduced another BASIC feature that controls the spacing. The TAB function used in each of the PRINT statements in Figure B-11 causes the typer to proceed to the column designated by the argument of the TAB function. The TAB function cannot be used to backspace the typer. If it has already passed the column called for by the argument of the TAB function, it is ignored. To minimize the number of spaces moved by the typer, semicolons denoting packed spacing should be used in lieu of commas in PRINT statements in which the TAB function appears.

User-defined functions may contain variables in addition to the variables used as the argument. For example, suppose R is related to T by the following equation:

$$R = a + bT + cT^2$$

A function corresponding to this relationship may be defined as follows:

```
LISTNH
10 DEF FNF(X)=(X+1)/(X+2+4*(X+1)+SQR(X))
20 PRINT "  X";TAB(14);"F(X)";TAB(26);"X";TAB(38);"F(X)";
30 PRINT TAB(50);"X";TAB(62);"F(X)"
40 FOR X=0 TO 48 STEP 4
50 PRINT X;TAB(9);FNF(X);
60 LET Y=X+50
70 PRINT TAB(24);Y;TAB(33);FNF(Y);
80 LET Z=X+100
90 PRINT TAB(48);Z;TAB(57);FNF(Z)
100 NEXT X
110 END
```

READY
RUNNH

X	F(X)	X	F(X)	X	F(X)
0	.25	50	1.88118E-2	100	9.69848E-3
4	.131579	54	1.74973E-2	104	9.33649E-3
8	8.75244E-2	58	1.63543E-2	108	9.00053E-3
12	6.51746E-2	62	1.53513E-2	112	8.68791E-3
16	5.18293E-2	66	1.44642E-2	116	8.39626E-3
20	4.29912E-2	70	1.36739E-2	120	8.12356E-3
24	3.67162E-2	74	1.29655E-2	124	.007868
28	3.20339E-2	78	1.23267E-2	128	7.62803E-3
32	2.84077E-2	82	.011748	132	7.40226E-3
36	2.55172E-2	86	1.12211E-2	136	7.18946E-3
40	2.31596E-2	90	1.07394E-2	140	6.98856E-3
44	2.12001E-2	94	1.02973E-2	144	6.79857E-3
48	1.95458E-2	98	9.89021E-3	148	6.61864E-3

Figure B-11 Example of the use of the function definition statement

DEF FNR(T) = A + T*(B + C * T)

When this function is executed, the current values of variables A, B, and C are used in computing the function.

B.16. SUBROUTINES

Basic's counterpart to the procedure described in Chapter 5 is a block of code (that is, the subroutine) within the program that is transferred to by the GOSUB statement whose format is

GOSUB *n*

where *n* is the line number of the first statement in the subroutine. In Basic, this statement is quite similar to the GOTO statement except that control can be transferred back to the first statement following the GOSUB statement via the statement

RETURN

This permits the subroutine to be "called," that is, a GOSUB statement executed, from several different locations in the remainder of the program.

As our first example of a subroutine in Basic, we shall prepare a routine that accepts a number (N), divides by another number (J) to see if that number is a factor of the other number, prints any factors found, and reduces the original number by dividing by all factors. This routine (lines 200-250) is used in Figure B-12 to prepare the Basic program for the flowchart in Figure 3-13. The GOSUB

```

LISTNH
10 FOR K=5780 TO 5790 STEP 2
20 PRINT "FACTORS OF",K
30 LET N=K
40 FOR J=2 TO N
50 GOSUB 200
60 NEXT J
70 NEXT K
80 STOP
200 LET L=INT(N/J)
210 IF L*J<>N THEN 250
220 PRINT J;" IS A FACTOR"
230 LET N=L
240 GOTO 200
250 RETURN
260 END

```

READY

RUNNH

```

FACTORS OF      5780
 2  IS A FACTOR
 2  IS A FACTOR
 5  IS A FACTOR
17  IS A FACTOR
17  IS A FACTOR
FACTORS OF      5782
 2  IS A FACTOR
 7  IS A FACTOR
 7  IS A FACTOR
59  IS A FACTOR
FACTORS OF      5784
 2  IS A FACTOR
 2  IS A FACTOR
 2  IS A FACTOR
 3  IS A FACTOR
241 IS A FACTOR
FACTORS OF      5786
 2  IS A FACTOR
11  IS A FACTOR
263 IS A FACTOR
FACTORS OF      5788
 2  IS A FACTOR
 2  IS A FACTOR
1447 IS A FACTOR
FACTORS OF      5790
 2  IS A FACTOR
 3  IS A FACTOR
 5  IS A FACTOR
193 IS A FACTOR

```

Figure B-12 Use of the subroutine feature in the factoring program

statement in line 50 transfers control to statement 200. The RETURN statement in line 250 directs control back to statement 60.

Although tempting, a GOTO or other similar transfer of control statement cannot be used to transfer from the subroutine to other parts of the program. Only the RETURN statement is used for this purpose. However, a subroutine may have more than one RETURN statement.

In essence, all variables in Basic are global. That is, all variables are accessible throughout the program.

As our second example of the use of a SUBROUTINE, we shall expand the stock-inventory flowchart in Figure 4-3 to include updating for items shipped as well as items received. Our program in Figure B-13 reads the initial inventory (lines 10-50), updates for items received (lines 60-90), updates for items shipped (lines 100-140), and then prints the final inventory (lines 150-180). This program uses a subroutine (lines 300-360) to locate the entry in array S that is equal to N and then increase the amount of the corresponding entry in array Q by amount R. For items shipped, R is set equal to -S in line 120 before transferring to the subroutine.

The subroutine in Figure B-13 has two RETURN statements (lines 340 and 360) and two GOSUB statements (lines 80 and 130). When the subroutine is called via the GOSUB statement in line 80, execution of either of the RETURN statements transfers control to line 90. When the subroutine is called via the GOSUB statement in line 130, execution of either of the RETURN statements transfers control to line 140.

As our final example of the use of a subroutine in Basic, we shall prepare the program corresponding to the enciphering process in Figure 4-17. The program in Figure B-14 uses a subroutine (lines 400-450) to determine the entry in the index table of Table 4-4b for a given character. At the time the subroutine is called, the character is stored in variable A\$, and upon return the corresponding index is stored in variable A. This program also has two GOSUB statements (lines 70 and 170). The subroutine has only one RETURN statement but also has a STOP statement.

Although not illustrated by our examples, a program may have several subroutines. It is also permissible for a subroutine to contain a GOSUB statement to call another subroutine.

B.17. IN SUMMARY

Although this appendix has hopefully covered the more common and frequently used features of Basic, it is by no means an exhaustive treatment. One important omission is statements pertaining to files. While they are generally mere extensions of the READ and PRINT statements, they tend to differ considerably from system to system. By this time, the reader should be able to consult manuals specific to his system to learn how to use files.

Basic also contains a variety of special statements called MAT statements for

460

```
LISTNH
10 DIM S(20),Q(20)
20 READ I
30 FOR J=1 TO I
40 READ S(J),Q(J)
50 NEXT J
60 READ N,R
70 IF N=0 THEN 100
80 GOSUB 300
90 GOTO 60
100 READ N,S
110 IF N=0 THEN 150
120 LET R=-S
130 GOSUB 300
140 GOTO 100
150 PRINT "ITEM","QUANTITY"
160 FOR J=1 TO I
170 PRINT S(J),Q(J)
180 NEXT J
190 STOP
300 FOR J=1 TO I
310 IF N=S(J) THEN 350
320 NEXT J
330 PRINT "NO ITEM NUMBER",N
340 RETURN
350 LET Q(J)=Q(J)+R
360 RETURN
400 REM INITIAL INVENTORY
410 DATA 8,1102,19,1543,21,1011,5,1917,8
420 DATA 1414,10,1998,3,2122,14,1201,8
430 REM UPDATES FOR ITEMS RECEIVED
440 DATA 1998,6,2122,5,1011,7,1534,9,1917,5
450 DATA 1102,8,1543,19,1998,3,1201,3,1102,5,0,0
460 REM UPDATES FOR ITEMS SHIPPED
470 DATA 1917,5,2122,8,1102,10,1011,3,1917,4
480 DATA 1102,5,1201,2,1102,8,1011,5,1998,6,0,0
490 END
```

READY
RUNNH

NO ITEM NUMBER	1534
ITEM	QUANTITY
1102	9
1543	40
1011	4
1917	4
1414	10
1998	6
2122	11
1201	9

Figure B-13 Use of the subroutine feature in the inventory program

performing matrix algebra. Since matrix operations have not been covered anywhere else in this text, discussion of these statements does not seem appropriate in this appendix. Anyone familiar with matrix algebra would find the MAT statements very straightforward.


```
LISTNH
10 DIM SS(26),KS(8),MS(16),CS(16),N(8)
20 FOR J=0 TO 26
30 READ SS(J)
40 NEXT J
50 FOR J=1 TO 8
60 READ KS(J)
70 NEXT J
80 FOR J=1 TO 16
90 READ MS(J)
100 NEXT J
110 FOR J=1 TO 8
120 LET AS=KS(J)
130 GOSUB 330
140 LET N(J)=A
150 NEXT J
160 LET J=0
170 FOR L=1 TO 16
180 LET J=J+1
190 IF J<>9 THEN 210
200 LET J=1
210 LET AS=MS(L)
220 GOSUB 330
230 LET M=A+N(J)+10
240 LET M=M-27*INT(M/27)
250 LET CS(L)=SS(M)
260 NEXT L
270 PRINT "CRYPTOGRAM IS",
280 FOR J=1 TO 16
290 PRINT CS(J);
300 NEXT J
310 STOP
330 FOR A=0 TO 26
340 IF SS(A)=AS THEN 380
350 NEXT A
360 PRINT "INVALID CHARACTER",AS
370 STOP
380 RETURN
400 REM DATA FOR ARRAY SS
410 DATA " ","A","B","C","D","E","F","G","H","I","J"
420 DATA "K","L","M","N","O","P","Q","R","S","T","U"
430 DATA "V","W","X","Y","Z"
440 REM KEYWORD
450 DATA "C","O","M","P","U","T","E","R"
460 REM MESSAGE
470 DATA "H","E"," " "H","A","D"," " "A"," "
480 DATA "B","A","D"," " "D","A","Y"
490 END
```

```
READY
RUNNH
CRYPTOGRAM IS  UCWGEGBM XCDGPZ
```

Figure B-14 Use of the subroutine feature in the enciphering program

Appendix



FORTRAN

The popularity of the Fortran language is apparently that it permits the programmer convenient access to most machine capabilities in a reasonably efficient manner. Fortran is relatively easy to learn, but the beginner must be cautioned that attention to details is absolutely necessary. Fortran has a wide variety of features, of which only the more important and commonly used are covered in this appendix. Unfortunately, Fortran is also somewhat machine dependent, although most of the features we will describe are fairly uniform in their implementation.

Statements in Fortran can be divided into five categories:

1. Arithmetic statements
2. Input/output statements
3. Control statements
4. Specification statements
5. Subprogram statements

While categorizing statements in this manner leads to a most orderly presentation, it is not best suited for someone trying to master the use of the language. We shall, therefore, not follow such an order but follow more closely the order in the first chapters of this text. This order should also permit the student to begin writing programs quite early, which is the only way to master a programming language.

C.1. ARITHMETIC ASSIGNMENT STATEMENTS

In Fortran, arithmetic assignment statements consist of constants, variables, operators, and functions.

Constants

In Fortran constants may assume one of three forms (See Section 6.3):

1. *Integer* (fixed-point) constants consist of a sign (optional) followed by a string of digits not containing either a decimal point or a comma. Valid examples include

124
100001
-102
+576

Unsigned integers are assumed positive. Invalid examples include

1,027 (contains a comma)
1.27 (contains a decimal point)
127. (contains a decimal point)
2/7 (contains an arithmetic operator, the slash)

Note that a decimal point following the string of digits is considered as part of the constant.

2. *Real* (floating-point) constants may be expressed in *decimal* format as a sign (optional) followed by a string of digits that contain a decimal point but no commas. These constants are stored internally in the real (floating-point) format. Valid examples include

1.276
-1027.
.00012
1270000.

Invalid examples include

1,270,000. (contains commas)
1027 (does not contain a decimal point)

Note that the difference between the constants 1 and 1. is significant. The constant 1 would be represented internally in integer form. The constant 1. would be represented internally in real (floating-point) form, with all the attendant characteristics such as nonexact representation in binary floating point (see Section 6.3).

3. Real constants may be expressed in *exponential* format consisting of the following sequence:
 - a. Sign of number (optional)
 - b. String of digits that contain a decimal point but no commas
 - c. The letter E
 - d. Sign of the exponent (optional)
 - e. String of digits that does not contain a decimal point

464 These constants are represented internally in the floating-point format. Valid examples include

1.276E - 14
0.00176 E + 4
9.765 E2
-1.765 E0

Invalid examples include

1.275 E 1.4	(exponent contains a decimal point)
156 E-2	(number preceding E does not contain a decimal point)
1.276 + 2	(does not contain the letter E)

Variables

Variable names in Fortran must conform to the following restrictions:

1. Generally, the only acceptable characters in a variable name are the twenty-six alphabetic characters, A thru Z, and the ten digits, 0 thru 9. A few systems accept the \$ in variable names.
2. The first character in the variable name must be an alphabetic character.
3. Most systems permit variable names to contain up to six characters although some systems only allow five.

In Fortran, a single storage location is reserved for each variable. The manner in which numerical data is stored in this location depends upon the name of the variable according to the following convention:

1. *Integer* variables begin with one of the letters I, J, K, L, M, or N. Valid examples of names for integer variables include

I
JACK
K27654

Invalid examples include

2I	(does not begin with an alphabetic character)
AI	(first character not I, J, K, L, M, or N)
JACKSON	(too many characters)
IA.J	(decimal point not acceptable within a variable name)

2. *Real* variables must begin with one of the letters A thru H or O thru Z. Valid examples of names for real variables include

F
HARRY

A10276
AI

Invalid examples include

- 2F (first character not alphabetic)
- HARRIET (too many characters)
- JASON (integer variable)
- H2 – B (minus sign not acceptable character in variable name)

Arithmetic Operators

Operators available in Fortran include the following:

Operation	Symbol	Example
Addition	+	A + B; I + J
Subtraction	–	A – B; I – J
Negation	–	–A; –I
Multiplication	*	A*B; I*J
Division	/	A/B; I/J
Exponentiation	**	A**B; I**J; A**I
Assignment	=	A=B: I=J; A=I: I=A

The examples shown are acceptable on all computers. Most current systems have removed the restrictions involved in the mixing of variable and constant types in expressions. For example, an expression such as

$$A + 2$$

is said to involve mixed-mode arithmetic, since *A* is a real variable whereas 2 is an integer constant. In processing this statement, the integer constant 2 would be converted to the real representation and then added to *A*.

In integer arithmetic, fractional parts resulting from divisions are truncated. That is, the result of the division 9/5 is 1. This is as if the function INT introduced in Section 2.5 were inserted into all divisions involving integer numbers.

The hierarchy of operations is as described in Section 2.7.

Unless the beginning programmer is very cautious, he will find many pitfalls connected with variable types. For example, suppose he wants to raise *A* to the one-third power. The following two representations are valid:

A**.33333
A**(1./3.)

Perhaps the second is trivially more accurate. However, the expression

$$A**(1/3)$$

466 does not produce the correct answer. The computer divides the integer 1 by the integer 3, obtaining a value of zero for the exponent.

There is also a subtle difference between the expressions

$$A^{**}2$$

and

$$A^{**}2.$$

which is probably not apparent to the beginner. The expression $A^{**}2$ is a real variable raised to an integer exponent. The expression $A^{**}2.$ is a real variable raised to a real exponent. Suppose you want to evaluate

$$9.6^{1.7}$$

How would you do it? Well, first you would take the logarithm

$$\ln [9.6^{1.7}] = 1.7 \ln 9.6$$

and then exponentiate:

$$e^{\ln [9.6^{1.7}] } = e^{1.7 \ln 9.6} = 9.6^{1.7}$$

The computer would treat the expression $A^{**}2.$ in a similar manner. That is, it would take the logarithm of A , multiply the result by 2., and then raise e to the resulting power. This is perfectly acceptable provided A is positive. When A is negative, the logarithm of A does not exist. Therefore, $A^{**}2.$ cannot be computed for negative values of A .

The expression $A^{**}2$ is effectively computed by multiplying A by itself, which can be accomplished for negative values of A , whereas $A^{**}2.$ is computable only for positive A . A secondary consideration is that $A^{**}2$ can be computed faster than can $A^{**}2.$

Finally, a word of wisdom to the beginning Fortran programmer. Pay special attention to the types of variables used, because herein lie many pitfalls. In many respects, this is the most difficult aspect of Fortran to master.

Functions

Fortran has a very comprehensive set of intrinsic functions. Perhaps the most commonly used subset is the following:

FUNCTION			
Integer	Real	Explanation	Example
IABS(J)	ABS(X)	Absolute value, x	ABS(A/2.); IABS(J*2)
	SQRT(X)	Square root, $\sqrt{x}$	SQRT(B**2-4.*A*C)

FUNCTION		Explanation	Example
Integer	Real		
	SIN(X)	Trigonometric sine, $\sin x$, x in radians	SIN (B/RAD)
	COS(X)	Trigonometric cosine, $\cos x$, x in radians	COS(B/RAD)
	ATAN(X)	Trigonometric arctangent, $\tan^{-1}(x)$ The result is in radians and located in quadrants as follows: $x > 0$ Quadrant I $x < 0$ Quadrant IV	ATAN((X+2.)/Z)
	ATAN2 (Y,X)	Trigonometric arctangent, $\tan^{-1}(y/x)$ The result is in radians and located in quadrants as follows: $x > 0, y > 0$ Quadrant I $x < 0, y > 0$ Quadrant II $x < 0, y < 0$ Quadrant III $x > 0, y < 0$ Quadrant IV	ATAN2(2.*R, -B)
	EXP(X)	The exponential function, e^x	EXP (A+B)
	ALOG(X)	The natural (base e) logarithm, $\ln x$	ALOG(-X*B)
	ALOG10(X)	The common (base 10) logarithm, $\log x$	ALOG10(X/2.)

Fortran permits very flexible use of its functions. In fact, any valid expression may be used as the argument of a function. The same or some other function may be used in the expression, as illustrated by the following examples:

```
X = ALOG(9. * ABS(Y/C))
G = SQRT(SQRT(B))
```

The last example computes the fourth root of B .

C.2. UNFORMATTED INPUT/OUTPUT

Most time-sharing systems and increasingly more batch-processing systems permit READ and PRINT statements to be used without an associated Format statement (described in the next section). For the beginning student, use of the unformatted input/output statements will make his first few programs considerably easier to prepare, provided, of course, that this option is open to him.

But since the Format feature provides substantially more flexibility in input/output, the student will probably find it necessary to master this feature before his study of Fortran is completed.

For input, the statement for reading values of A, B, and J would be as follows:

```
READ, A, B, J
```

On some time-sharing systems, the statement for entry of data at the terminal is

```
ACCEPT, A, B, J
```

The numerical values of A, B, and J would then be entered in that order via the appropriate input device, normally the teletype for time-sharing systems or the card reader for batch-processing units. The numerical values are normally separated either by a comma or by one or more blank spaces. For example, the input data

```
1.717, 3.546E2, -5
```

would assign the value 1.717 to A, 3.546E2 to B, and -5 to J. Note that the numerical values to be assigned to real variables are real constants in either the decimal or exponential format and that values assigned to integer variables are in the integer format. Mixing is often not allowed.

The fact that a blank space can be used to separate values means that a number may not have any embedded blanks. In the case of card input, however, this permits multiple cards to be used for the input data. The computer scans the first card and accepts whatever values are given on it. If more values are needed, it automatically proceeds to the next card. That is, the numerical values in the previous paragraph could have just as well been presented as

```
1.717, 3.546E2
```

on the first card and

```
-5
```

on the second card. The values can be positioned anywhere on the card.

Processing of a READ statement automatically advances the input scan to the next record, that is, next line or next card. That is, the input data for the three READ statements

```
READ, A
READ, B
READ, J
```

must be the values

```
1.717
3.546E2
-5
```

each on a separate line or card.

For output, the appropriate statement is the PRINT statement. For example, to output values for A, B, and J, the statement would be

```
PRINT, A, B, J
```

For the numerical values in the previous paragraph, this statement would generate the following type of output:

```
0.1717000 E 01      0.3546000 E 03      -5
```

Real variables are generally printed in the exponential format, although some systems will substitute the decimal format for numbers between 0.1 and 10^8 . Integer variables are always printed in integer format.

The output for each PRINT statement starts on a new line. That is, the statements

```
PRINT, A
PRINT, B
PRINT, J
```

would give the values on three successive lines. If more variables are specified in a given print statement than can be printed on a single line, the overflow is continued on the next line.

Descriptive material can be readily incorporated into output data. For example, the following statement

```
PRINT, 'A = ', A
```

prints

```
A = 0.1717000 E 01
```

Similarly, the statement

```
PRINT, A, 'IS THE ANSWER'
```

prints

```
0.1717000 E 01 IS THE ANSWER
```

The statement

```
PRINT, ' '
```

prints a blank line.

Many systems use the quotation mark symbol " in place of the apostrophe that was used in the above examples and that will be used throughout this appendix.

C.3. FORMATTED INPUT/OUTPUT

The use of Format-free input/output is quite acceptable provided (a) the feature is available on your computer and (b) the output format you want conforms

with that produced by the format-free convention. If either of these does not hold, formatted input/output must be used.

For formatted input/output, the READ statement assumes one of the following two forms:

```
READ (j, n) A, B, J
READ n, B,
```

where j is the unit number for the input device (usually 5 for the card reader), and n is the statement number of the associated Format statement. For output, the appropriate statement is either of the following

```
PRINT n, A, B, J
WRITE (j, n) A, B, J
```

where j is the unit number of the output device (usually 6 for the line printer), and n is the statement number of the associated Format statement.

The format statement assumes the following form:

```
n FORMAT (f1, f2, f3, . . .)
```

where n is its statement number.

$f_1, f_2, f_3, \dots$ are the field specifications for input/output.

In this section we shall cover only five fields but shall introduce some additional features in later sections.

I-Field

To enter integer values, the appropriate field is the I-field, whose field specification is rIw , where r is the repetition number and w is the width of the field. For example, consider the following READ statement:

```
READ 14, J, K, L
```

to be used on a computer with card input. Suppose we use the following Format statement:

```
14 FORMAT (3I5)
```

This statement says that there are three fields on the card, each of which is five characters in width. The first field always starts with column 1 and the following fields appear consecutively on the card. Therefore, the value of J will be obtained from columns 1-5, the value of K from columns 6-10, and the value of L from columns 11-15.

If the values 11, -6, and 27 are to be entered for J , K , and L , respectively, the input should appear as in Figure C-1a. Note that the numbers appear in the rightmost columns in their respective fields; that is, they are *right-justified*. A good rule to remember is that, on most systems, blanks in a numeric input field are interpreted as zeros. Thus, the input shown in Figure C-1b would assign the values 11000, -6000, and 27000 to J , K , and L , respectively. This same rule, that is, blanks are equivalent to zeros, also applies to other numeric fields.

As further examples of Format statements, the input in Figure C-1c would be appropriate for the following Format statement:

```
14 FORMAT (I2, 2I3)
```

When the repetition number is 1, as for the first field, it is omitted. The input in Figure C-1d would be appropriate for the Format statement:

```
14 FORMAT (I2, I3, I4)
```

On output, the number is always right-justified. Negative values are printed with a sign; positive numbers are not preceded by a sign. In computing the width w , a column must be reserved for the minus sign. For example, field I3 would print numbers between -99 and 999.

F-Field

One of the two fields that can be used to enter real values is the F-field, whose general form is $rFw.d$, where r is the repetition number, w is the width of the field, and d is the number of digits to the right of the decimal point. For example, consider the following READ statement:

```
READ 24, A, B, C
```

to be used for card input. Suppose we use the following Format statement:

```
24 FORMAT (3F6.0)
```

If a decimal point will always be provided with the input data, the d in the field specification is ignored and can always be made zero. Figure C-2a gives an appropriate card for entering the values 1.27, 15.1, and 0.027 for A, B, and C respectively. While the numbers within the fields in Figure C-2a are right-justified, this is not necessary as illustrated in Figure C-2b. Again, blanks in numeric fields are treated as zeros. Thus, the input for the first field in Figure C-2b is

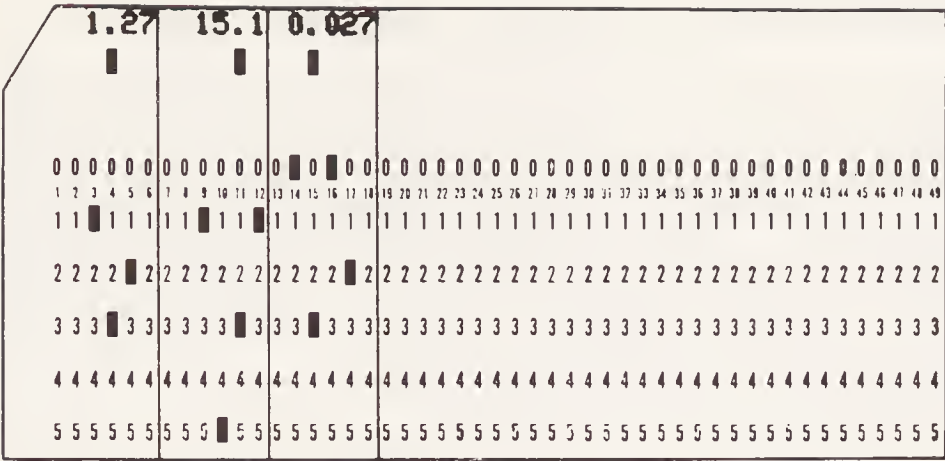
```
1.2700
```

Since the zeros are trailing zeros following a decimal point, they have no effect on the numerical values.

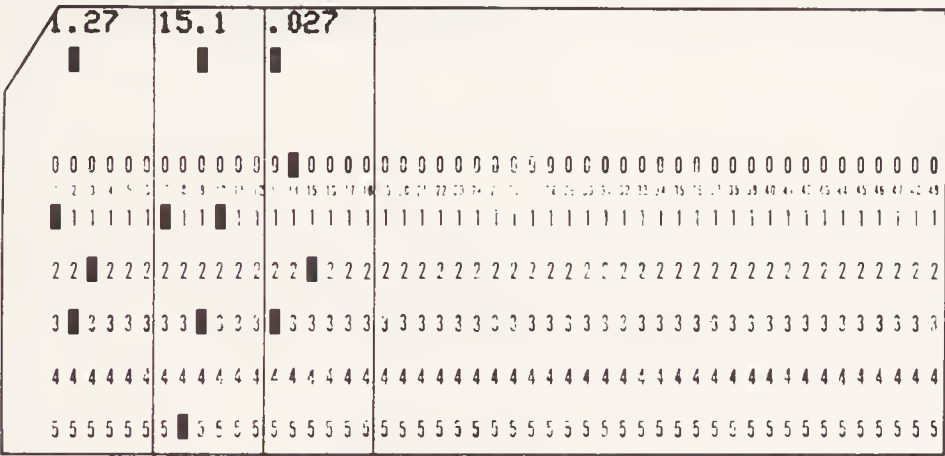
Again, different width fields can be used if desirable. For example, the input in Figure C-2c would be appropriate for the Format statement

```
24 FORMAT (F4.0, F5.0, F6.0)
```

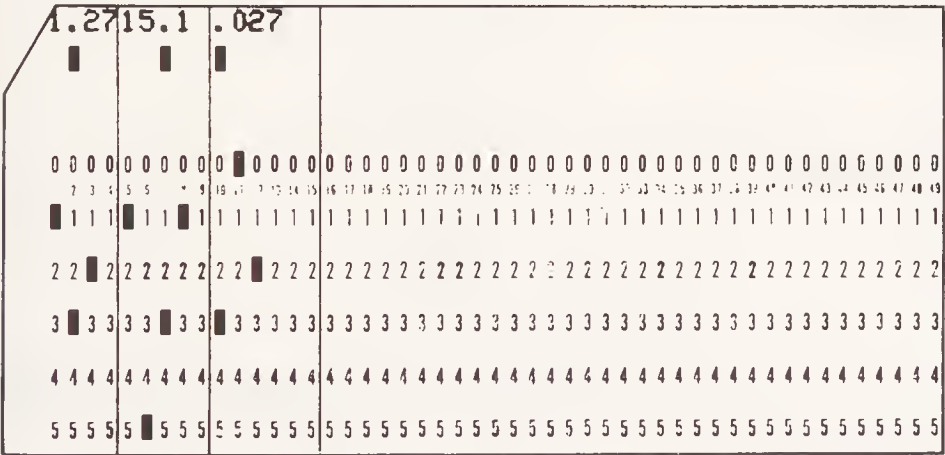
On output, the value is always right-justified in the field, and d signifies the number of digits to the right of the decimal point. For example, the number 15.12 printed under field F8.3 would be written 15.120 and right-justified. On output, the decimal point is always provided and must be considered when choosing the field width w . Similarly, for negative numbers, a column is occupied by the minus sign. Positive numbers on output are written without signs. For example, the field F6.2 is large enough to accommodate numbers between -99.99 and 999.99.



a. Input data for 3F6.0



b. Alternate input data for 3F6.0



c. Input data for F4.0, F5.0, F6.0

Figure C-2 Input data for F-fields

On output, the rightmost digit is rounded. That is, printing 87.267 under F 6.2 gives output 87.27.

E-Field

Real numbers can be read or printed in exponential format via use of the E-field, whose general field specification in *rEw.d*, where *r* is the repetition number, *w* is the field width, and *d* is the number of digits to the right of the decimal point. For example, the READ statement

```
READ 24, A, B, C
```

might have the format statement

24 FORMAT (3E10.0)

Again, if the decimal point is always provided on input, the d part of the field specification has no effect and can be set to zero. Figure C-3a gives the proper data card to enter -12700., 15.2, and 0.00122. The numbers are right justified within the fields, which is generally necessary for E-fields. For example, suppose we use the input in Figure C-3b. Since blanks are treated as zeros, the first number becomes 1.27E40000, which is not what is desired.

On output, the number is typically written as follows:

$$s0 \cdot X_1 X_2 \cdots X_d E s Y Y$$

where s is a sign field that contains a minus sign for negative values or is blank for positive values. For example, the number 27.34 written under E-format would appear as ($d=4$)

$$0.2734E\ 02$$

right-justified in the field.

In computing the field width, both signs, the leading zero, the decimal point, and the letter E must be considered. The minimum field width is

	Characters
Sign of number	1
Leading zero	1
Decimal point	1
Digits	d
Letter E	1
Sign of exponent	1
Exponent	2
Total	$d+7$

Some systems insert a blank preceding the E, so to be conservative, the minimum field width should be $w \geq d + 8$.

X-Field

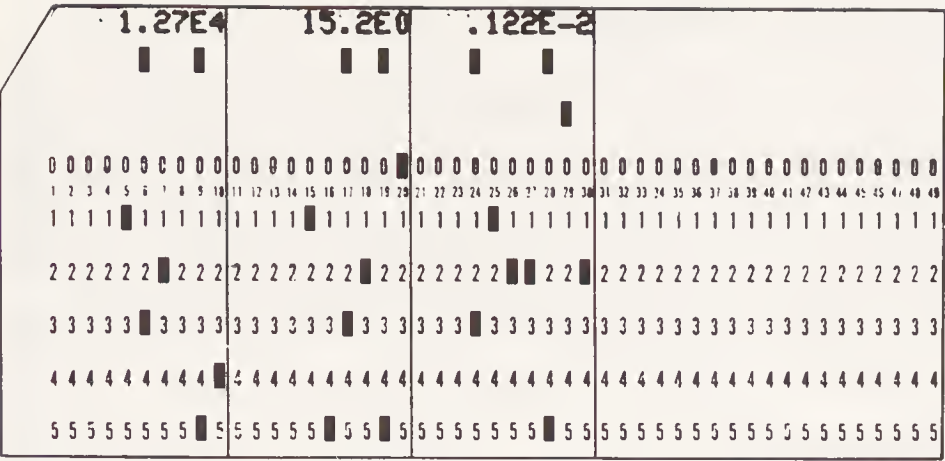
The X-field (specification rX) is used to skip characters in the input data or to insert blanks in the output. For example, on input, the Format statement

$$19\ \text{FORMAT}\ (I5, 4X, F6.0)$$

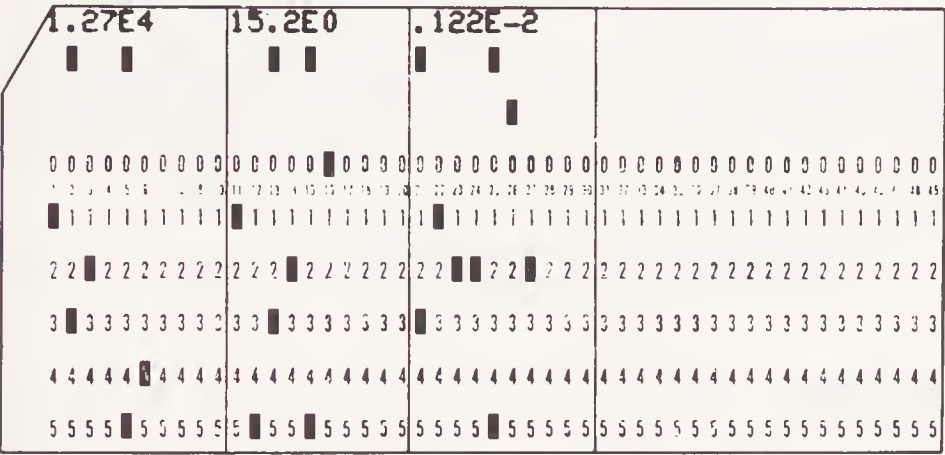
would extract an integer number from columns 1 thru 5, completely ignore whatever is in columns 6 thru 9, and extract a real number from columns 10 thru 15.

Hollerith Field

To insert explanatory material into the output, the Hollerith field specification can be used. For example, the statements



a. Input data for 3E10.0



b. Invalid input for 3E10.0

Figure C-3 Input data for E-fields

```
A = 4.2716
PRINT 2,A
2 FORMAT ('□TOTAL□IS', F6.2)
```

would print.

TOTAL □IS□□4.27

where □ represents a blank character.

To use the Hollerith field, the characters to be inserted into the output line are simply enclosed in apostrophes (or in quotation marks on some systems).

In fact, no variables have to appear in the PRINT statement. That is, the following statements are perfectly valid:

```
PRINT 20
20 FORMAT ('□OUTPUT□DATA')
```

Some systems do not permit use of the apostrophe or quotation marks in Format statements. Instead, they use the H-field as illustrated in the following examples (equivalent to those presented above):

```
2 FORMAT (9H□TOTAL□IS, F6.2)
20 FORMAT (12H□OUTPUT□DATA)
```

The number of characters appears prior to the H in this field specification. When available, the apostrophe or quotation marks are more convenient.

Special Condition for Line Printer Output

In the Format statement used with any output statement for a line printer, the beginning programmer should always be sure that the first character in the output line is a blank character. Note that in each of the examples in the paragraphs on the Hollerith field, a blank character appeared at the start of the Hollerith field. The same effect can be obtained with the X-field. For example, the statements

```
PRINT 15, A, B, C
15 FORMAT (1X, F5.2, E17.5, F8.1)
```

use the 1X field at the start of the FORMAT statement to insure that the first character is a blank.

The reason for this is that the line printer does not print the first character but uses it to set the spacing between lines, that is, for carriage control. A blank character specifies single spacing. We shall return to this subject later.

Teletype users need not be concerned with carriage control.

C.4. A COMPLETE PROGRAM

In this section, we shall discuss the remaining concepts that will enable us to write a complete program.

First, we shall note that the statements in a Fortran program can be divided into two categories: *executable* statements and *nonexecutable* statements. Executable statements are those that specify action and include arithmetic statements, READ statements, and PRINT statements. Nonexecutable statements are those that supply information for the computer or information to be used in executing an executable statement. For example, the FORMAT statement is a nonexecutable statement, since it only supplies information needed in order to execute a READ or PRINT statement.

Execution of a Fortran program always begins with the first executable statement. Termination of the program is via either the STOP statement or by the CALL EXIT statement. In current systems, either of these statements transfers control from the user's program back to the supervisor or executive system that oversees operation of the computer. These statements are executable in that they call for action, namely, the termination of the execution of the program. Most programs will have at least one of these statements but may have more than one.

In compiling a program, the Fortran compiler begins with the very first statement in the program and continues until an END statement is encountered. This statement is always the very last statement in the program, and its only function is to inform the compiler that the end of the program has been reached. Therefore, this is a nonexecutable statement, and the program must contain only one END statement.

Figure C-4 gives two complete examples of a Fortran program. One of them is for Format-free; the other uses Format. The example is taken from Exercise 2-22, which entails the computation of the minimum number of coins in the change for purchases under \$1.00. Only pennies, nickels, dimes, and quarters are used, and the purchaser always submits a dollar bill.

The program uses integer arithmetic throughout. The amount of the purchase in cents is read, and then subtracted from 100 to give the total number of pennies if only they are given in change. The next statements then convert any multiples of 25 to quarters, the next statements convert any multiples of 10 in the remaining pennies to dimes, and similarly for nickels.

Flow of execution is from executable statement to executable statement until a STOP statement is encountered. Figure C-4b illustrates that FORMAT statements may be placed anywhere within the program; that is, they need not follow immediately after the READ, PRINT, or WRITE statement.

C.5. PREPARATION OF PROGRAM

The manner in which a Fortran program is to be punched onto cards has become highly standardized. While time-sharing systems tend to be similar, minor variations exist from one system to the next. We shall therefore first describe the layout for cards and then consider time-sharing systems.

Figure C-5 illustrates the coding form commonly used for punching Fortran programs. The use of the columns is as follows:

- Columns 7 thru 72.* The Fortran statement is punched within these columns.

- Columns 1 thru 5.* Statement numbers are punched in these columns. Blanks are ignored, so the statement number may appear anywhere within these columns.

- Columns 73 and 80.* Ignored by the Fortran compiler.

- Column 6.* If column 6 contains any character except a blank or zero, the contents of columns 7 thru 72 are treated as a continuation of the contents of columns 7 thru 72 of the previous card.

- Column 1* If column 1 contains a C, the card is treated as a comment card, and its contents ignored by the Fortran compiler. This feature permits the programmer to insert explanatory comments into his program.

Note specifically that this applies only to statements contained in the Fortran

```

          $JOB          1105.50003
1         READ, IPUR
2         IPEN=100-IPUR
3         IQUAR=IPEN/25
4         IPEN=IPEN-IQUAR*25
5         IDIME=IPEN/10
6         IPEN=IPEN-IDIME*10
7         NICK=IPEN/5
8         IPEN=IPEN-5*NICK
9         PRINT, 'CHANGE IS'
10        PRINT, IQUAR, ' QUARTERS'
11        PRINT, IDIME, ' DICES'
12        PRINT, NICK, ' NICKELS'
13        PRINT, IPEN, ' PENNIES'
14        STOP
15        END

```

```

          $ENTRY
CHANGE IS
          2  QUARTERS
          1  DICES
          0  NICKELS
          1  PENNIES

```

a. Format-free version

```

          $JOB          1105.50003
1         READ(5,10) IPUR
2         10 FORMAT(15)
3         IPEN=100-IPUR
4         IQUAR=IPEN/25
5         IPEN=IPEN-IQUAR*25
6         IDIME=IPEN/10
7         IPEN=IPEN-IDIME*10
8         NICK=IPEN/5
9         IPEN=IPEN-5*NICK
10        WRITE(6,20)
11        WRITE(6,30) IQUAR
12        WRITE(6,40) IDIME
13        WRITE(6,50) NICK
14        WRITE(6,60) IPEN
15        STOP
16        20 FORMAT(' CHANGE IS')
17        30 FORMAT(1X,15,' QUARTERS')
18        40 FORMAT(1X,15,' DICES')
19        50 FORMAT(1X,15,' NICKELS')
20        60 FORMAT(1X,15,' PENNIES')
21        END

```

```

          $ENTRY
CHANGE IS
          2  QUARTERS
          1  DICES
          0  NICKELS
          1  PENNIES

```

b. Program with format

program itself. That is, *it does not apply to data cards*. Data can start in column 1 and extend through column 80 as desired.

Figure C-6 contains the complete punched deck corresponding to the program in Figure C-4a. This deck has been assembled for submission to the WATFIV (*Waterloo Fortran IV*) compiler. In addition to the cards for the Fortran program itself and the cards for the data, a typical deck must also contain *control cards* that give instructions to the supervisor or executive program. The very first card in this deck begins with \$JOB and contains the user's charge number beginning in column 16. The Fortran program then follows this card. Between the program and data is the \$ENTRY control card. The \$JOB control card signifies the start of a new program to be compiled. The program is read and compiled in response to this card. The \$ENTRY card instructs the computer to execute the program that has just been compiled. The data cards are read during the execution phase and thus follow the \$ENTRY card.

Unfortunately, control cards are generally specific to the computer utilized and are often further tailored to meet the needs of the specific installation. Therefore, the above discussion will generally be useful only as an example of control cards.

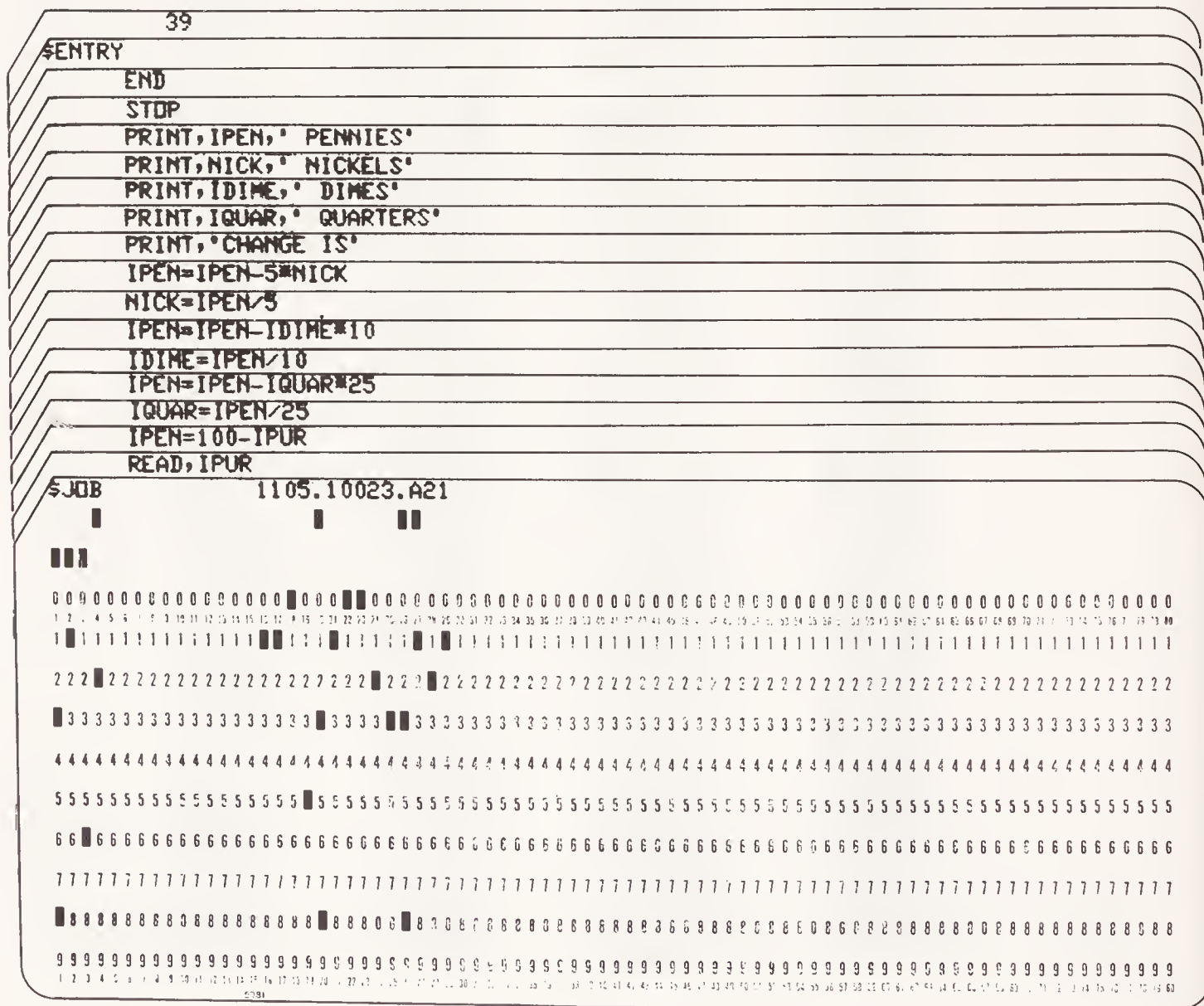


Figure C-6 Punched-card deck

For time-sharing systems, the program is generally submitted one line at a time. Each line generally begins with a line number, usually consisting of five digits or fewer. Some systems require the following characters to conform precisely to the format given above for cards. Others relax the requirements considerably. For example, a typical requirement is that the line number be followed by a space, then by the statement number (if it exists) and another space, and then by the statement itself. Examples of two time-sharing runs are given in Figure C-7, again using the program in Figure C-4. The formats are

LIST

SMITH 11:34 THUR 03/23/72

```

10 INPUT,IPIR
20 IPEN=100-IPIR
30 IQUAR=IPEN/25
40 IPEN=IPEN-IQUAR*25
50 IDIME=IPEN/10
60 IPEN=IPEN-IDIME*10
70 NICK=IPEN/5
80 IPEN=IPEN-NICK*5
90 PRINT 20
100 PRINT 30,IQUAR
110 PRINT 40,IDIME
120 PRINT 50,NICK
130 PRINT 60,IPEN
140 STOP
150 20 FORMAT("CHANGE IS")
160 30 FORMAT(I5," QUARTERS")
170 40 FORMAT(I5," DIMES")
180 50 FORMAT(I5," NICKELS")
190 60 FORMAT(I5," PENNIES")
200 END

```

READY
RUN

SMITH 11:35 THUR 03/23/72

INPUT:00010
? 39

```

CHANGE IS
  2 QUARTERS
  1 DIMES
  0 NICKELS
  1 PENNIES

```

STOP

a. G-E time-sharing system

Figure C-7 Examples of time-sharing output

482

```

00010      READ(5,10) IPUR
00020      10  FORMAT(I5)
00030      IPEN=100-IPUR
00040      IOUAR=IPEN/25
00050      IPEN=IPEN-IOUAR*25
00060      IDIME=IPEN/10
00070      IPEN=IPEN-IDIME*10
00080      NICK=IPEN/5
00090      IPEN=IPEN-NICK*5
00100      WRITE(6,20)
00110      WRITE(6,30) IOUAR
00120      WRITE(6,40) IDIME
00130      WRITE(6,50) NICK
00140      WRITE(6,60) IPEN
00150      STOP
00160      20  FORMAT(' CHANGE IS')
00170      30  FORMAT(1X,I5,' QUARTERS')
00180      40  FORMAT(1X,I5,' DIMES')
00190      50  FORMAT(1X,I5,' NICKELS')
00200      60  FORMAT(1X,I5,' PENNIES')
00210      END
END OF DATA
run
G1 COMPILER ENTERED
SOURCE ANALYZED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
39
CHANGE IS
    2 QUARTERS
    1 DIMES
    0 NICKELS
    1 PENNIES
EDIT
end
NOTHING SAVED
ENTER SAVE OR END-
end
READY
logoff

```

b. IBM time-sharing system

Figure C-7 (Continued)

substantially different, so again instructions specific to the installation are necessary.

C-6. TRANSFER OF CONTROL

Fortran offers six types of statements that effect a transfer of control:

1. Unconditional GOTO
2. Logical IF
3. Arithmetic IF
4. Computed GOTO
5. Assigned GOTO
6. END = option in READ statements

We shall describe all except the fifth one.

These types of statements are used to cause the program to execute some statement other than the next one in order. In Fortran, any executable state-

ment can be given a statement number, and transfer of control statements can direct control to any executable statement that has a statement number.

Unconditional GOTO

The format of this statement is

```
GOTO n
```

and is used to direct control to the executable statement whose statement number is *n*. A positive integer constant must be used for *n*; variables are not allowed.

Logical IF

The format of the logical IF is as follows:

```
IF (relational expression) statement
```

where *statement* can be any of the executable statements we have covered so far except that it cannot be another IF statement, and *relational expression* is a logical expression whose value is either TRUE or FALSE. If the relational expression is TRUE, *statement* is executed. If relational expression is FALSE, the next statement executed is the one immediately following the IF statement. When *statement* is executed and it is not a transfer of control statement, that is, not a GOTO statement, the next statement executed is the one following the IF statement.

The formulation of the relational expression is analogous to that indicated in Section 2.4. Fortran recognizes the following six relational operators:

Operator	Explanation
.LT.	Less than
.LE.	Less than or equal
.GT.	Greater than
.GE.	Greater than or equal
.EQ.	Equal
.NE.	Not equal

Note that the relational operator consists of four characters, the first and last of which are decimal points.

As an example of the use of transfer of control statements, Figure C-8 gives the Fortran programs equivalent to the flowchart in Figure 2-12. Note the use of two logical IF statements in this program. In each case, *statement* is a GOTO statement, but this is not a restriction on the logical IF. Also note the use of two STOP statements.

Fortran also permits the construction of a compound conditional much along the lines discussed in Section 2.8. The following logical operators are recognized:

484

```

$JOB                                1105.5^003
1      M=313791
2      IX=M/2
3      N=2
4      19 J=M/N
5      IF(J*N.EQ.M)GO TO 7
6      N=N+1
7      IF(N.GT.IX)GO TO 19
8      PRINT,M,' IS A PRIME NUMBER'
9      STCP
10     7 PRINT,M,' IS NOT A PRIME NUMBER'
11     STOP
12     END
```

```

$ENTRY
313791 IS A PRIME NUMBER
```

a. Format-free version

```

$JOB                                1105.50003
1      M=313791
2      IX=M/2
3      N=2
4      19 J=M/N
5      IF(J*N.EQ.M)GO TO 7
6      N=N+1
7      IF(N.GT.IX)GO TO 19
8      WRITE(6,10)M
9      STOP
10     7 WRITE(6,11)M
11     STOP
12     10 FORMAT(1X,I6,' IS A PRIME NUMBER')
13     11 FORMAT(1X,I6,' IS NOT A PRIME NUMBER')
14     END
```

```

$ENTRY
313791 IS A PRIME NUMBER
```

b. Version with format

Figure C-8 Fortran implementation of prime-number algorithm of Figure 2-12

Operator	Explanation
.AND.	Logical AND
.OR.	Logical OR
.NOT.	Logical NOT

Again note that the first and last characters of the logical operators are decimal points.

To illustrate the use of multiple conditionals in Fortran, Figure C-9 gives the Fortran counterpart to the flowchart in Figure 2-15b. Note that the *statement* portion of this logical IF is an arithmetic statement.


```

$JOB      1105.50003
1      READ,X
2      FX=X**2
3      IF(X.LT.-1..OR.X.GT.1.)FX=1.
4      PRINT,'F(X)=',FX
5      STOP
6      END

```

```

$ENTRY
F(X)=      0.9715453E 00

```

a. Format-free version

```

$JOB      1105.50003
1      READ(5,1)X
2      FX=X**2
3      IF(X.LT.-1..OR.X.GT.1.)FX=1.
4      WRITE(6,2)FX
5      STOP
6      1  FORMAT(F10.5)
7      2  FORMAT(1X,'F(X)=',F7.5)
8      END

```

```

$ENTRY
F(X)=0.97155

```

b. Version with format

Figure C-9 Use of compound conditionals

Arithmetic IF

The arithmetic IF provides another means by which conditional branching can be obtained. The format of this statement is

IF (*arithmetic expression*) n_1, n_2, n_3

where *arithmetic expression* is any valid arithmetic expression, and

n_1, n_2 , and n_3 are statement numbers of executable statements.

If the numerical value of the *arithmetic expression* is negative, control is transferred to n_1 ; if the value is zero, control is transferred to n_2 ; if positive, control is transferred to n_3 . Three statement numbers must always be given, although two of them may be identical. As in the GOTO statement, n_1, n_2 , and n_3 must be positive integer constants; variables are not allowed.

The programs in Figure C-10 are equivalent to those in Figure C-8 except that arithmetic IF's are used in place of logical IF's.

Computed GOTO

The computed GOTO is Fortran's equivalent to multiple branching as described in Section 3.6. The format of the computed GOTO statement is

GOTO ($n_1, n_2, \dots, n_k$), *integer*

486

```

$JOB                                1105.50003

1      M=313791
2      IX=M/2
3      N=2
4      19 J=M/N
5      IF(J*N-M)16,7,16
6      16 N=N+1
7      IF(N-IX)28,28,19
8      28 PRINT,M,' IS A PRIME NUMBER'
9      STOP
10     7 PRINT,M,' IS NOT A PRIME NUMBER'
11     STOP
12     END
```

```

$ENTRY
313791 IS A PRIME NUMBER
```

a. Format-free version

```

$JOB                                1105.50003

1      M=313791
2      IX=M/2
3      N=2
4      19 J=M/N
5      IF(J*N-M)16,7,16
6      16 N=N+1
7      IF(N-IX)28,28,19
8      28 WRITE(6,10)M
9      STOP
10     7 WRITE(6,11)M
11     STOP
12     10 FORMAT(1X,I6,' IS A PRIME NUMBER')
13     11 FORMAT(1X,I6,' IS NOT A PRIME NUMBER')
14     END
```

```

$ENTRY
313791 IS A PRIME NUMBER
```

b. Version with format

Figure C-10 Use of arithmetic IF

when $n_1, n_2, \dots, n_k$ are statement numbers and *integer* is an integer variable. If *integer* equals 1, control is transferred to n_1 , if *integer* equals 2, control is transferred to n_2 , etc. An example of the use of this statement is presented in the next paragraph.

END = Option in the READ Statement

The general format of the READ statement is as follows:

```

READ (j, n1, END = n2, ERR = n3) list
```

- where j = unit number of the input device
- n_1 = statement number of the FORMAT statement to be used
- n_2 = statement number to which control is to be transferred upon encounter of an end-of-data

n_3 = statement number to which control is to be transferred upon encounter of error in the input data

If not wanted, the END = or ERR = option can be omitted. When format-free input is being used, systems such as WATFIV use an * in place of n_1 .

Figure C-11 gives the Fortran program corresponding to the flowchart in Figure 3-17b. This program illustrates the use of the computed GOTO and the END = feature in the READ statement. Not all systems recognize the END = and ERR = options in the READ statement as presented in this section, but most systems provide some approach to accomplish the same objective.

C.7. THE DO STATEMENT

The function of the iteration counter described in Chapter 3 is provided by the DO statement in Fortran. The format of this statement is as follows:

DO *n integer* = m_1, m_2, m_3

```

$JOB                                1105.50003

1      JAGE=0
2      JF=0
3      JSP=0
4      JUN=0
5      JSR=0
6      50 READ(5,*,END=100)IAGE, ISTND
7      JAGE=JAGE+IAGE
8      GO TO (10,20,30,40),ISTND
9      10 JF=JF+1
10     GO TO 50
11     20 JSP=JSP+1
12     GO TO 50
13     30 JUN=JUN+1
14     GO TO 50
15     40 JSR=JSR+1
16     GO TO 50
17     100 STUD=JF+JSP+JUN+JSR
18     AVG=JAGE/STUD
19     PFRESH=JF*100./STUD
20     PSOPH=JSP*100./STUD
21     PJUN=JUN*100./STUD
22     PSEN=JSR*100./STUD
23     PRINT,AVG
24     PRINT,PFRESH,PSOPH,PJUN,PSEN
25     STOP
26     END

$ENTRY
0.2000000E 02
0.2500000E 02      0.3750000E 02      0.2500000E 02      0.1250000E 02

```

a. Format-free version

Figure C-11 Fortran implementation of flowchart in Figure 3-17b

488

```

$JOB                                1105.50003

1      JAGE=0
2      JF=0
3      JSP=0
4      JUN=0
5      JSR=0
6      50 READ(5,60,END=100) IAGE,ISTND
7      JAGE=JAGE+IAGE
8      GO TO (10,20,30,40),ISTND
9      10 JF=JF+1
10     GO TO 50
11     20 JSP=JSP+1
12     GO TO 50
13     30 JUN=JUN+1
14     GO TO 50
15     40 JSR=JSR+1
16     GO TO 50
17     100 STUD=JF+JSP+JUN+JSR
18     AVG=JAGE/STUD
19     PFRESH=JF*100./STUD
20     PSOPH=JSP*100./STUD
21     PJUN=JUN*100./STUD
22     PSEN=JSR*100./STUD
23     WRITE(6,61)AVG
24     WRITE(6,62)PFRESH,PSOPH,PJUN,PSEN
25     STOP
26     60 FORMAT(9X,I2,4X,I1)
27     61 FORMAT(1X,F10.3)
28     62 FORMAT(1X,4F10.3)
29     END
```

\$ENTRY				
20.000				
25.000	37.500	25.000	12.500	

b. Version with format

18	1
19	2
17	1
21	3
22	4
19	3
24	2
20	2

c. Input data

Figure C-11 (Continued)

where n is the statement number of the last statement in the loop constructed by the iteration counter

integer is an integer variable to be used as the index of the loop

m_1 integer variable or constant whose value is assigned to *integer* upon entry into the loop

m_2 integer variable or constant that whenever exceeded by *integer* results in an exit from the loop

m_3 integer variable or constant by which integer is incremented for each pass around the loop

Some versions of Fortran permit expressions to be used for m_1 , m_2 , and m_3 . If the value of m_3 is 1, it can be omitted from the DO statement. Virtually all versions of Fortran require that m_3 be positive, but more and more versions permit m_1 and m_2 to be zero or negative.

In Fortran, the DO loop is constructed as per Version I in Figure 3-7. Upon exit, control is transferred to the first executable statement following the DO loop.

As our first example of the use of the DO statement in Fortran, Figure C-12 gives the program corresponding to the flowchart in Figure 3-9 for computing an entry in the Fibonacci sequence. The iteration loop consists of the statements starting with the DO statement and continuing through statement number 27, which is the end of the *range* of the DO loop. After executing the loop the desired number of times, the next statement to be executed is the one following statement number 27.

```

$JOB 1105,50003
1 JENT1=1
2 JENT2=0
3 READ,NENT
4 DO 27J=3,NENT
5 JENT=JENT1+JENT2
6 JENT2=JENT1
7 27 JENT1=JENT
8 PRINT,'ENTRY',NENT,' IS ',JENT
9 STOP
10 END

```

```

$ENTRY
ENTRY 31 IS 832040

```

a. Format-free version

```

$JOB 1105,50003
1 JENT1=1
2 JENT2=0
3 READ(5,10)NENT
4 DO 27J=3,NENT
5 JENT=JENT1+JENT2
6 JENT2=JENT1
7 27 JENT1=JENT
8 WRITE(6,11)NENT,JENT
9 STOP
10 10 FORMAT(I7)
11 11 FORMAT(1X,'ENTRY',I3,' IS ',I7)
12 END

```

```

$ENTRY
ENTRY 31 IS 832040

```

b. Version with format

Figure C-12 Computation of an entry in the Fibonacci sequence in Fortran

In some cases it is necessary to use a dummy or no-action statement as the last statement in the range of the DO. In Fortran, this statement is the CONTINUE statement. An example of its use is given in Figure C-13, which is the Fortran counterpart to the flowchart in Figure 3-11.

In Figure 3-11, the “no” arm of the decision block calls for the index to be incremented and the loop repeated. In Fortran, this is accomplished by transferring to the end of the range of the DO loop, never back to the DO statement itself. Transfer of control to the DO statement is permissible, but this causes the index to be reinitialized, not incremented. Since no operations are required between the decision block and incrementing the index, we can only insert the CONTINUE statement at the end of the DO loop, as illustrated in Figure C-13.

One discrepancy between the program in Figure C-13 and the flowchart in Figure 3-11 is the use of NUM instead of NUMBER as an indexing parameter.

```

$JOB 1105.50003
1 READ,NUMBER
2 NUM=NUMBER
3 DO 70 J=2,NUM
4 60 K=NUMBER/J
5 IF(J*K.NE.NUMBER)GO TO 70
6 PRINT,J,' IS A FACTOR'
7 NUMBER=K
8 GO TO 60
9 70 CONTINUE
10 STOP
11 END

$ENTRY
2 IS A FACTOR
14327 IS A FACTOR
```

a. Format-free version

```

$JOB 1105.50003
1 READ(5,10)NUMBER
2 NUM=NUMBER
3 DO 70 J=2,NUM
4 60 K=NUMBER/J
5 IF(J*K.NE.NUMBER)GO TO 70
6 WRITE(6,11)J
7 NUMBER=K
8 GO TO 60
9 70 CONTINUE
10 STOP
11 10 FORMAT(I5)
12 11 FORMAT(1X,I5,' IS A FACTOR')
13 END

$ENTRY
2 IS A FACTOR
14327 IS A FACTOR
```

b. Version with format

Figure C-13 Computation of factors

Some versions of Fortran do not permit the indexing parameters to be modified within the range of the DO loop. WATFIV is one of these. In these cases, the only solution is to use a counter in place of the DO statement.

Nested DO loops are permitted in Fortran, as illustrated by the program in Figure C-14, which is the counterpart to the flowchart in Figure 3-13. In

```

      $JOB          1105.50003
1      DO 80 IA=5780,5790,2
2      NUMBER=IA
3      NUM=NUMBER
4      PRINT,'FACTORS OF',NUMBER
5      DO 70 J=2,NUM
6      60 K=NUMBER/J
7      IF(J*K,NE,NUMBER)GO TO 70
8      PRINT,J,' IS A FACTOR'
9      NUMBER=K
10     GO TO 60
11     70 CONTINUE
12     80 CONTINUE
13     STOP
14     END

```

```

      $ENTRY
FACTORS OF          5780
      2 IS A FACTOR
      2 IS A FACTOR
      5 IS A FACTOR
     17 IS A FACTOR
     17 IS A FACTOR
FACTORS OF          5782
      2 IS A FACTOR
      7 IS A FACTOR
      7 IS A FACTOR
     59 IS A FACTOR
FACTORS OF          5784
      2 IS A FACTOR
      2 IS A FACTOR
      2 IS A FACTOR
      3 IS A FACTOR
     241 IS A FACTOR
FACTORS OF          5786
      2 IS A FACTOR
     11 IS A FACTOR
     263 IS A FACTOR
FACTORS OF          5788
      2 IS A FACTOR
      2 IS A FACTOR
    1447 IS A FACTOR
FACTORS OF          5790
      2 IS A FACTOR
      3 IS A FACTOR
      5 IS A FACTOR
     193 IS A FACTOR

```

a. Format-free version

Figure C-14 Use of nested DO loops

```

$JOB                                1105.5003
1      DO 80 IA=5780,5790,2
2      NUMBER=IA
3      NUM=NUMBER
4      WRITE(6,10)NUMBER
5      DO 70 J=2,NUM
6      60 K=NUMBER/J
7      IF(J*K.NE.NUMBER)GO TO 73
8      WRITE(6,11)J
9      NUMBER=K
10     GO TO 63
11     70 CONTINUE
12     80 CONTINUE
13     STOP
14     10 FORMAT(' FACTORS OF',15)
15     11 FORMAT(1X,15,' IS A FACTOR')
16     END

```

```

$ENTRY
FACTORS OF 5780
2 IS A FACTOR
2 IS A FACTOR
5 IS A FACTOR
17 IS A FACTOR
17 IS A FACTOR
FACTORS OF 5782
2 IS A FACTOR
7 IS A FACTOR
7 IS A FACTOR
59 IS A FACTOR
FACTORS OF 5784
2 IS A FACTOR
2 IS A FACTOR
2 IS A FACTOR
3 IS A FACTOR
241 IS A FACTOR
FACTORS OF 5786
2 IS A FACTOR
11 IS A FACTOR
263 IS A FACTOR
FACTORS OF 5788
2 IS A FACTOR
2 IS A FACTOR
1447 IS A FACTOR
FACTORS OF 5790
2 IS A FACTOR
3 IS A FACTOR
5 IS A FACTOR
193 IS A FACTOR

```

b. Version with format

Figure C-14 (Continued)

Fortran, the index of the outer DO must not be the same as the index for the inner DO. Similarly, the inner DO must lie entirely within the range of the outer DO. As discussed in Section 3.4, their ranges must never overlap.

C.8. SPECIFICATION STATEMENTS

The most commonly used specification statements in Fortran are the INTEGER, REAL, DIMENSION, and COMMON statements. One use of the INTEGER and REAL statements is to alter the normal convention that variables whose name begins with one of the alphabetic characters I thru N are treated as integers and others as real. For example, the statement

```
REAL MU, JACK, KELLY
```

causes variables MU, JACK, and KELLY to be treated as real. Similarly, the statement

```
INTEGER ALPHA, X, GAMMA
```

causes variables ALPHA, X, and GAMMA to be treated as integers.

Another use of the specification statements is to allocate storage to arrays. The DIMENSION statement allocates such storage without changing the normal convention of the naming of integer and real. For example, the statement

```
DIMENSION AR(16), JK(5), ALPHA(3,6)
```

defines AR to be a real array of sixteen elements, JK as an integer array of five elements, and ALPHA as a double-subscripted array of eighteen elements. Fortran does not reserve a location for element zero.

The COMMON statement can also reserve storage for arrays but will not be covered until a later section.

The REAL and INTEGER statements can also be used to allocate storage to arrays. For example, the statement

```
REAL A(15), K(6)
```

defines arrays A and K to be real arrays containing fifteen and six elements, respectively. Similarly, the statement

```
INTEGER A(15), K(6)
```

would define these as integer arrays.

Fortran does not permit the same variable name to be defined in more than one way. For example, if A is defined as a single-subscripted real array, it cannot be used as a simple variable.

Most programmers place specification statements prior to the first executable statement. Some systems require this; others are not so rigid.

C.9. SUBSCRIPTED VARIABLES

In Fortran, subscripts are enclosed in parentheses and follow the variable names. Subscripts are restricted to integers, but they may be integer constants, integer variables, or integer expressions. Before a subscripted variable may be used, it

494

```

$JOB                                1105.50003
1      DIMENSION IA(10)
2      SUM=0.
3      DO 10 J=1,7
4      READ,IA(J)
5      10 SUM=SUM+IA(J)
6      AVG=SUM/7.
7      PRINT,'AVERAGE AGE IS',AVG
8      DO 20 J=1,7
9      DIF=IA(J)-AVG
10     20 PRINT,J,IA(J),DIF
11     STOP
12     END

$ENTRY
AVERAGE AGE IS      0.3585713E 02
      1      21      -0.1485713E 02
      2      36      0.1428680E 00
      3      45      0.9142868E 01
      4      64      0.2814287E 02
      5      25      -0.1085713E 02
      6      21      -0.1485713E 02
      7      39      0.3142868E 01
```

a. Format-free version

```

$JOB                                1105.50003
1      DIMENSION IA(10)
2      SUM=0.
3      DO 10 J=1,7
4      READ(5,25)IA(J)
5      10 SUM=SUM+IA(J)
6      AVG=SUM/7.
7      WRITE(6,30)AVG
8      DO 20 J=1,7
9      DIF=IA(J)-AVG
10     20 WRITE(6,35)J,IA(J),DIF
11     STOP
12     25 FORMAT(6X,I2)
13     30 FORMAT(1X,'AVERAGE AGE IS ',F5.2)
14     35 FORMAT(1X,I2,3X,I2,3X,F6.2)
15     END

$ENTRY
AVERAGE AGE IS 35.86
 1  21  -14.86
 2  36   0.14
 3  45   9.14
 4  64  28.14
 5  25 -10.86
 6  21 -14.86
 7  39   3.14
```

b. Version with format

Figure C-15 Use of arrays in Fortran

must have appeared in a specification statement, either DIMENSION, INTEGER, or REAL.

Figure C-15 gives the Fortran program that corresponds to the flowchart in

Figure 4-2. Although only seven storage locations will be needed, ten locations are reserved for the integer array IA for the ages. The first DO loop reads the ages into array IA and sums them. The second DO loop prints the ages and their differences from average.

Figures C-16a and b give the Fortran program that corresponds to the flowchart in Figure 4-3. Since the stock number and quantity in inventory will both be integers, the INTEGER statement is used to define these arrays as integer. The remainder of the program closely parallels the flowchart in Figure 4-3.

Fortran also has a special input/output statement for arrays called the *implied DO loop*, which is very similar to that described in Section 4.5. Figures C-16c and d contain examples of the use of this feature. In Figure C-16c, the second statement

```
READ, ITEMS, (STOCK (J), QUAN (J), J=1, ITEMS)
```

is interpreted as follows:

1. Read a value for ITEMS
2. Read values for STOCK (1) and QUAN (1)
3. Read values for STOCK (2) and QUAN (2)
4. Continue this process for consecutive values of J until J equals ITEMS.

With format-free I/O, the location of the values on the cards is immaterial as long as the values appear in the proper order.

To use this same feature in Figure C-16d, several new format features must be

```

$JOB      1105.50003
1         INTEGER STOCK(20),QUAN(20),STKNO
2         READ, ITEMS
3         DO 25 J=1, ITEMS
4         25 READ,STOCK(J),QUAN(J)
5         40 READ(5,*,END=50)STKNO,NREC
6         DO 30 J=1, ITEMS
7         IF(STKNO.EQ.STOCK(J))GO TO 35
8         30 CONTINUE
9         PRINT,'NO ITEM NUMBER',STKNO
10        GO TO 40
11        35 QUAN(J)=QUAN(J)+NREC
12        GO TO 40
13        50 DO 45 J=1, ITEMS
14        45 PRINT,STOCK(J),QUAN(J)
15        STOP
16        END

$ENTRY
NO ITEM NUMBER      1985
1108                250
1068                245
1972                210
1066                200
1984                200

```

a. Format-free version

Figure C-16 Inventory problem in Fortran

```

$JOB 1105,50003
1      INTEGER STOCK(20),QUAN(20),STKNO
2      READ(5,46) ITEMS
3      DO 25 J=1, ITEMS
4      25 READ(5,47) STOCK(J),QUAN(J)
5      40 READ(5,47,END=50) STKNO,NREC
6      DO 30 J=1, ITEMS
7      IF(STKNO.EQ.STOCK(J))GO TO 35
8      30 CONTINUE
9      WRITE(6,48) STKNO
10     GO TO 40
11     35 QUAN(J)=QUAN(J)+NREC
12     GO TO 40
13     50 DO 45 J=1, ITEMS
14     45 WRITE(6,49) STOCK(J),QUAN(J)
15     STOP
16     46 FORMAT(I7)
17     47 FORMAT(6X,I4,5X,I3)
18     48 FORMAT(' NC ITEM NUMBER ',I4)
19     49 FORMAT(1X,I4,2X,I4)
20     END

```

```

$ENTRY
NC ITEM NUMBER 1985
1108 250
1068 245
1972 210
1066 200
1984 200

```

b. Version with format

```

$JOB 1105,50003
1      INTEGER STOCK(20),QUAN(20),STKNO
2      READ, ITEMS, (STOCK(J),QUAN(J),J=1, ITEMS)
3      40 READ(5,*,END=50) STKNO,NREC
4      DO 30 J=1, ITEMS
5      IF(STKNO.EQ.STOCK(J))GO TO 35
6      30 CONTINUE
7      PRINT,'NO ITEM NUMBER',STKNO
8      GO TO 40
9      35 QUAN(J)=QUAN(J)+NREC
10     GO TO 40
11     50 DO 45 J=1, ITEMS
12     45 PRINT,STOCK(J),QUAN(J)
13     STOP
14     END

```

```

$ENTRY
NO ITEM NUMBER 1985
1108 250
1068 245
1972 210
1066 200
1984 200

```

c. Format-free version with implied DO

Figure C-16 (Continued)


```

$JOB      1105.50003

1      INTEGER STOCK(20),QUAN(20),STKNO
2      READ(5,47) ITEMS,(STOCK(J),QUAN(J),J=1,ITEMS)
3      40 READ(5,46,END=50)STKNO,NREC
4      DO 30 J=1,ITEMS
5      IF(STKNO.EQ.STOCK(J))GO TO 35
6      30 CONTINUE
7      WRITE(6,48)STKNO
8      GO TO 40
9      35 QUAN(J)=QUAN(J)+NREC
10     GO TO 40
11     50 WRITE(6,49)(STOCK(J),QUAN(J),J=1,ITEMS)
12     STOP
13     46 FORMAT(I10,I8)
14     47 FORMAT(I7/(I10,I8))
15     48 FORMAT(' NO ITEM NUMBER ',I4)
16     49 FORMAT(1X,I4,2X,I4)
17     END

```

\$ENTRY	
NO ITEM NUMBER 1985	
1108	250
1068	245
1972	210
1066	200
1984	200

d. Version with format with implied DO

5	
1108	150
1068	245
1972	110
1066	100
1984	200
1972	100
1108	100
1066	100
1985	021

e. Input data for all versions

Figure C-16 (Continued)

introduced. The first is the slash (/), which says to terminate scanning the current record and proceed to the next. Second, when the scan proceeds through the end of the format statement but there remain variables to be read, the scan reverts back to the first open parenthesis and proceeds to the next record. For example, the statements

```

      READ 47, ITEMS, (STOCK (J), QUAN(J), J= 1,ITEMS)
      47 FORMAT (I7/(I10,I8))

```

are interpreted as follows:

1. Read the value of ITEMS from the first seven columns of the first card.

- 2. The slash causes the scan to proceed to the next card.
- 3. The values of STOCK(1) and QUAN(1) are taken from columns 1-10 and 11-18, respectively, of this card.
- 4. This brings us to the end of the format statement. The scan now reverts back to the first open parenthesis, that is, the one preceding the field I10. The next value entered will also be taken from the next card.
- 5. Values of STOCK (2) and QUAN (2) are read, and the process continues.

When using format, the implied DO feature can also be used in the output statement:

```
PRINT 49, (STOCK (J), QUAN (J), J = 1, ITEMS)
49 FORMAT (IX, I4,2X,I4)
```

The values of STOCK (1) and QUAN (1) are written on the first line, which brings us to the end of the field specifications in the format statement. Control now reverts to the first open parenthesis, which is the one following the word FORMAT, and the next output will be on the next line. STOCK (2) and QUAN (2) are then written on the second line, and the process continues.

C.10. CARRIAGE CONTROL FOR THE LINE PRINTER

When using format, a carriage control character must be provided for each line of output to the line printer. This character is always the very first character in the output line, and is never printed. The following characters are recognized by practically all systems:

Character	Result
blank	Output begins on next line.
0	Output begins on the second line; i.e., one line is skipped.
1	Output begins on the first line at the top of the next page.
+	Output begins without advancing the carriage; i.e., an overprint is obtained.

Not only must carriage control be provided at the beginning of every format statement used for output but also after every slash and after an open parenthesis to which control would revert when the end of the format field specifications is reached.

C-11. THE DATA STATEMENT

Suppose in a Fortran program the value -27 is to be stored in variable JACK. Up to this point, we could use either the arithmetic assignment statement

JACK = -27

or the READ statement

READ, JACK

where the corresponding data card contained the entry -27. While either of these could be used, they both are executable statements which require execution time.

The DATA statement provides an alternate approach by instructing the compiler to insert the value -27 into the storage location reserved for JACK. The DATA statement for this example would be as follows:

DATA JACK/-27/

The word DATA is followed by the variable or variables to be initialized, which are in turn followed by the corresponding values enclosed in slashes.

The following example illustrates the initialization of three variables:

DATA A,B,J/1.2, -1.7E4, 14/

Real and integer variables may be mixed, but the value corresponding to a real variable must be expressed as a real constant and the value corresponding to an integer variable must be an integer constant.

The DATA statement is especially attractive for initializing arrays, as illustrated by the following example:

DIMENSION A(14)
DATA A/1., 13*0./

This statement causes 1. to be stored in A(1) and zero to be stored in the remainder of the array. Note the use of the repetition feature for the zero entries. Most versions of Fortran require that the entire array be initialized; that is, it is not possible to selectively initialize certain elements but not others.

The DATA statement is not an executable statement. Therefore, it should not be given a statement number nor can it be used to restore variables to their values at the beginning of the program. It is certainly possible for the value of a variable that appears in a DATA statement to be changed later in the program.

Examples of the use of the DATA statement appear in the next section.

C.12. NONNUMERIC DATA

To enter nonnumeric data via the format feature, the field specification is rAw , where r is the repetition number and w is the width of the field. In general, the maximum value for w is limited to the maximum number of characters that can be stored in a single storage location. For systems like the IBM/360, this value is four.

Figure C-17 illustrates a program that reads a temperature in either degrees Fahrenheit or degrees centigrade and prints the temperature on both scales. The


```

1  $JOB                                1105.50003
2  DATA ICHF, ICHC/'F','C' /
3  READ(5,1)T, ICH
4  FORMAT(F10.0,A1)
5  IF(ICH.EQ. ICHC)GOTO10
6  IF(ICH.EQ. ICHF)GOTO20
7  WRITE(6,2) ICH
8  FORMAT(' INVALID CHARACTER ',A1,' ENTERED')
9  GOTO40
10 DC=T
11 DF=1.8*DC+32.
12 GOTO30
13 DF=T
14 DC=(DF-32.)/1.8
15 WRITE(6,3)DC,DF
16 FORMAT(' TEMPERATURE IS',F7.2,' DEG C OR',F7.2,' DEG F')
17 GOTO40
18 END
```

```

$ENTRY
TEMPERATURE IS 28.17 DEG C OR 82.70 DEG F
TEMPERATURE IS 142.50 DEG C OR 288.50 DEG F
INVALID CHARACTER G ENTERED
```

a. Program with output

Figure C-17 Temperature conversion program

```

82.7      F
142.5     C
152.3     G

```

b. Input data

Figure C-17 (Continued)

value is entered in the first ten columns of a data card followed by either an F or a C in column 11 to indicate which scale the reading is in. The two IF statements following the READ statement compare the entered character with the characters F and C, which were stored in variables ICHF and ICHC via the DATA statement. If the character is neither an F nor a C, an error message is printed. Otherwise, the necessary conversion is made and the answers printed.

The DATA statement can be used to store character data in variables in much the same fashion as it is used to store numerical data. The characters to be stored are simply enclosed in apostrophes. Many systems also permit the use of the H-field, as indicated by the following example:

```
DATA ICHF, ICHC/1HF,1HC/
```

The number of characters to be stored in a single storage location is still limited to the maximum as pointed out at the beginning of this section.

Although character data may be stored in either integer or real variables, integer is generally preferred. In any case, the programmer should be consistent in his use throughout the program. If careless, an undesired arithmetic conversion of integer to real or vice versa may be encountered. For example, consider the following two statements:

```
DATA ICHA,CHA / 'A', 'A' /
IF (ICA.EQ.CHA) GOTO7
```

The computer will not conclude that ICHA and CHA are the same. The IF statement is executed as if ICHA and CHA both contain numerical quantities. Since ICHA is integer and CHA is real, ICHA will be converted arithmetically from integer to real before the comparison is made, and the result will not equal CHA.

To read strings whose number of characters exceeds the number that can be stored in a single storage location, arrays can be conveniently used. For example, to read a string of 20 characters, the following statements can be used:

```

DIMENSION K(5)
READ 26, (K(I), I=1, 5)
26 FORMAT (5A4)

```

The READ statement specifies that the entire array consisting of five storage locations is to be read. The FORMAT statement specifies that there are five consecutive fields of four characters in width, giving a total of twenty characters.

```
$JOB      1105,50003
1  INTEGER SYM(27),MSG(16),NKEY(8),CRYP(16),KEY(8)
2  DATA SYM/ ' ','A','B','C','D','E','F','G','H','I','J','K','L','M',
3  1 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',/
4  READ(5,1)(KEY(J),J=1,8),(MSG(J),J=1,16)
5  DO 2 J=1,8
6  DO 3 N=1,27
7  IF(KEY(J).EQ.SYM(N)) GO TO 2
8  3 CONTINUE
9  WRITE(6,5)KEY(J)
10 STOP
11 2 NKEY(J)=N-1
12 J=0
13 DO 6 L=1,16
14 J=J+1
15 IF(J.EQ.9)J=1
16 DO 7 N=1,27
17 IF(MSG(L).EQ.SYM(N)) GO TO 8
18 7 CONTINUE
19 WRITE(6,9)MSG(L)
20 STOP
21 8 M=N-1+NKEY(J)+10
22 M=M-27*(M/27)
23 CRYP(L)=SYM(M+1)
24 6 CONTINUE
25 WRITE(6,10)(CRYP(L),L=1,16)
26 STOP
27 1 FORMAT(8A1/16A1)
28 5 FORMAT('OINVALID CHARACTER',A1,'IN KEYWORD')
29 9 FORMAT('OINVALID CHARACTER',A1,'IN MESSAGE')
30 10 FORMAT('OCRYPTOGRAM',5X,16A1)
    END
```

\$ENTRY
CRYPTOGRAM UCWGEGOBM XCDGPZ
a. Program with output

Figure C-18 Cryptogram program

COMPUTER
HE HAD A BAD DAY
b. Input data

Figure C-18 (Continued)

The DATA statement can also be used to initialize arrays containing character data, as illustrated by the following example:

```
DIMENSION MONTH (2)  
DATA MONTH/'DECEMBER'/
```

To illustrate the processing of nonnumeric data, our first example will be the cryptogram problem whose flowchart is given in Figure 4-17. The program in Figure C-18 parallels this flowchart very closely except for the fact that Fortran does not reserve a storage location for the zero subscript. In defining the index table in Table 4-4b, the index has been advanced by 1 so that the result can be stored in array SYM via the DATA statement in the second line of the program in Figure C-18.

Whereas the indexing of the loops for K in Figure 4-17 is from 0 to 26, the indexing of the corresponding loops in Figure C-18 is from 1 to 27.

Our second example of the processing of nonnumeric data will be a program corresponding to the flowchart in Figure 4-18. The program in Figure C-19 corresponds almost directly to the flowchart. Variables ICA, ICB, ICN, ICT, ICH, and ICE are assigned the character A, blank, N, T, H, and E, respectively, by the DATA statement. Some versions of Fortran will accept statements of the type

```
IF(C(J).NE.'A') GOTO50
```

In these cases, use of the DATA statement could be avoided.

C.13. FUNCTIONS AND SUBPROGRAMS

Fortran has three types of features relating to functions and subprograms. These are:

1. The statement function
2. The function subprogram
3. The subroutine

We shall treat each of these separately but with this point of introduction. The statement function is a part of the program in which it appears and is compiled as part of that program. The function subprogram and subroutine are separate routines and are compiled as an entity. That is, each has its own structure, its own declaration statements, its own END statement, etc. In many cases, the function subprogram or subroutine is compiled immediately after the main program in which their call appears, but the compiler carries over absolutely no

```

$JOB      1105,50003
1      INTEGER L(50),C(50)
2      DATA ICA,ICB,ICN,ICT,ICH,ICE/'A',' ','N','T','H','E'/
3      READ(5,1)(C(J),J=1,37)
4      1 FORMAT(37A1)
5      DO 2 J=1,36
6      2 L(J)=J+1
7      L(37)=0
8      ITXT=1
9      IFREE=0
10     NTXT=0
11     NFREE=0
12     JO=1
13     10 J=JO
14     IF(C(J).NE.ICA)GO TO 50
15     J=L(J)
16     IF(C(J).EQ.ICB)GO TO 30
17     IF(C(J).NE.ICN)GO TO 80
18     J=L(J)
19     IF(C(J).NE.ICB)GO TO 80
20     30 IF(IFREE.EQ.0)GO TO 31
21     L(NFREE)=JC
22     GO TO 32
23     31 IFREE=JO
24     32 IF(NTXT.EQ.0)GO TO 33
25     L(NTXT)=L(J)
26     GO TO 34
27     33 ITXT=L(J)
28     34 NTXT=J
29     NFREE=J
30     JO=L(J)
31     L(J)=0
32     GO TO 10
33     50 IF(C(J).NE.ICT)GO TO 80
34     J=L(J)
35     IF(C(J).NE.ICH)GO TO 80
36     J=L(J)
37     IF(C(J).NE.ICE)GO TO 80
38     J=L(J)
39     IF(C(J).EQ.ICB)GO TO 30
40     80 J=L(J)
41     IF(J.EQ.0)GO TO 100
42     IF(C(J).NE.ICB)GO TO 80
43     NTXT=J
44     JO=L(J)
45     GO TO 10
46     100 J=ITXT
47     105 WRITE(6,110)C(J)
48     J=L(J)
49     IF(J.NE.0)GO TO 105
50     110 FORMAT(10X,A1)
51     STOP
52     END

```

\$ENTRY

A
P
P
L
E
D
A
Y

K
E
E
P
S

D
O
C
T
O
R

A
W
A
Y
•

a . Program with output

Figure C-19 Fortran implementation of the flowchart in Figure 4-18

AN APPLE A DAY KEEPS THE DOCTOR AWAY.

b. Input data

Figure C-19 (Continued)

information concerning variables, statements, variable types, etc., to the subprogram compilation. This produces the advantage that duplicate statement numbers and duplicate variable names (but in no way associated) may be used between the main program and the subprogram, but also produces the disadvantage that declaration statements that apply to the subprogram must be repeated.

Statement Function

The statement function is restricted to a function that can be defined by a single Fortran statement. A good example is a sine function for which the argument is in degrees rather than radians. Such a function, say, `SINDG`, can be defined as follows:

$$\text{SINDG}(X) = \text{SIN} (X / 57.3)$$

The variable `X` in this statement is a dummy variable; that is, no storage location is reserved for it. The variable `X` can be used again later on in the program. To compute the trigonometric sine of `ANGLE`, which is in degrees, the appropriate statement is

$$S = \text{SINDG}(\text{ANGLE})$$

That is, the statement function is very convenient for defining new functions for use within a given program.

The statement function may have several arguments; for example,

$$\text{HYP} (A,B) = \text{SQRT} (A**2 + B**2)$$

Also, the statement function may contain variables in addition to those appearing as arguments. For example, suppose the electrical resistance is a function of temperature as follows:

$$R = a + bT + cT^2$$

The statement function could be

$$\text{RES} (T) = A + T*(B + C*T)$$

Only variable `T` is a dummy. When `RES` is called, it is evaluated using the current values of `A`, `B`, and `C`.

A good example of the use of the statement function is in the program in Figure C-20 for printing the tabular function in Figure 4-10. As programmed in Figure C-20, the program corresponds to the flowchart in Figure 4-11b. The statement function appears as the first statement in the program. The statement function is itself not an executable statement in that it only defines the function. Thus, execution begins with the `PRINT` statement in the second line.

Most Fortran systems require that statement functions appear prior to any executable statements. In Figure C-20, the variable X is the dummy variable in the function definition. Later in the program, variable X is again used, but there is no connection between that variable X and the dummy variable X in the function definition.

```

      $JOB                                1105.50003
1      F(X)=(X+1.)/(X**2+4.*(X+1.))+SQRT(X))
2      WRITE(6,1)
3      1 FORMAT('1',3(7X,'X',6X'F(X)'))
4      X=0.
5      3 IF(X.GT.48.) STOP
6      FX=F(X)
7      Y=X+50.
8      FY=F(Y)
9      Z=X+100.
10     FZ=F(Z)
11     WRITE(6,2)X,FX,Y,FY,Z,FZ
12     2 FORMAT(1X,3(F8.1,F10.5))
13     X=X+2.
14     GO TO 3
15     END
```

\$ENTRY

a. Program

X	F(X)	X	F(X)	X	F(X)
0.0	0.25000	50.0	0.01881	100.0	0.00970
2.0	0.17227	52.0	0.01813	102.0	0.00951
4.0	0.13158	54.0	0.01750	104.0	0.00934
6.0	0.10534	56.0	0.01691	106.0	0.00917
8.0	0.08752	58.0	0.01635	108.0	0.00900
10.0	0.07475	60.0	0.01584	110.0	0.00884
12.0	0.06517	62.0	0.01535	112.0	0.00869
14.0	0.05775	64.0	0.01489	114.0	0.00854
16.0	0.05183	66.0	0.01446	116.0	0.00840
18.0	0.04700	68.0	0.01406	118.0	0.00826
20.0	0.04299	70.0	0.01367	120.0	0.00812
22.0	0.03961	72.0	0.01331	122.0	0.00799
24.0	0.03672	74.0	0.01297	124.0	0.00787
26.0	0.03422	76.0	0.01264	126.0	0.00775
28.0	0.03203	78.0	0.01233	128.0	0.00763
30.0	0.03011	80.0	0.01203	130.0	0.00751
32.0	0.02841	82.0	0.01175	132.0	0.00740
34.0	0.02689	84.0	0.01148	134.0	0.00729
36.0	0.02552	86.0	0.01122	136.0	0.00719
38.0	0.02428	88.0	0.01097	138.0	0.00709
40.0	0.02316	90.0	0.01074	140.0	0.00699
42.0	0.02214	92.0	0.01051	142.0	0.00689
44.0	0.02120	94.0	0.01030	144.0	0.00680
46.0	0.02034	96.0	0.01009	146.0	0.00671
48.0	0.01955	98.0	0.00989	148.0	0.00662

b. Output data

Figure C-20 Program to print tabular function

The program in Figure C-20 also illustrates the use of a repetition number for groups of fields within a Format statement. For example, in statement 2, the notation 3(F8.1,F10.5) is equivalent to F8.1, F10.5, F8.1, F10.5, F8.1, F10.5. The repetition feature is also used in statement 1 of Figure C-20.

Function Subprogram

The function subprogram in Fortran is a complete program very similar to the main program except that it begins with the FUNCTION statement and contains at least one RETURN statement. Figure C-21 gives the function subprogram corresponding to the flowchart in Figure 5-4. The first statement names this function as SQROOT and indicates that there will be one argument, namely, X. In Fortran, the value returned is always that of the variable in the subprogram whose name is identical to the name of the function, SQROOT in this case.

On many systems, the transfer is via the “call by address” as described in Section 5.3. That is, the address of the variable used as the argument in the calling program is transferred to the subprogram, which does not establish a separate storage location for its argument, thereby making it a dummy. However, other systems use “call by value” when the respective argument is a simple variable. Manuals specific to the system being used must be consulted on this point.

```

      $JOB                1105.50003

1      DO 8 J=1,2
2      READ,X
3      DELTA=SQROOT(X)
4      8 PRINT,DELTA
5      STOP
6      END

7      FUNCTION SQROOT(X)
8      G=1.
9      10 H=X/G
10     IF(ABS(G-H).LT.0.001)GO TO 20
11     G=(H+G)*0.5
12     GO TO 10
13     20 SQROOT=G
14     RETURN
15     END

      $ENTRY
0.5000023E 01
0.3872983E 01

```

a. Program with output

25.0

15.0

b. Input data

Figure C-21 Example of a function subprogram

Figure C-22 illustrates the use of a function subprogram in the program corresponding to the flowchart in Figure 3-13. Basically, the function subprogram IFCT accepts a number to be factored, NUMBER, and a proposed factor, J, determines if J is a factor of NUMBER, prints the factors, divides NUMBER by any factors found, and returns the result to the main program.

Note the use of statement numbers 1 and 2 in both programs. Also note the use of variable K in both programs. The main program and the function subprogram are compiled separately. In compiling the subprogram, the compiler reserves storage locations for all variables except those appearing in the argument list. Therefore, a storage location is reserved for K in the subprogram and is a different storage location than the one reserved for K in the main program.

In choosing a name for a function, consideration must be given as to whether the value associated with the function is integer or real. Since the function name is used as a variable name in the subprogram, the usual naming convention is followed. For the function subprogram in Figure C-22, the value associated with the function name is integer, and thus a name beginning with one of the letters I through N was chosen.

The INTEGER and REAL declarations can be used to alter the naming convention if desired, as illustrated in Figure C-23. Note the use of the word INTEGER in the FUNCTION statement in the function subprogram. Also note the use of the INTEGER statement in the main program to declare the name FACTOR as integer. An INTEGER statement for FACTOR in the subprogram is not needed since FACTOR is defined as integer for the subprogram in the FUNCTION statement.

The REAL declaration can be used in an analogous manner.

As our next example of the use of a function subprogram, note that the flowchart in Figure 4-17 for the enciphering process calls for the look-up of the index table in Table 4-4b twice, once for the characters in the keyword and once for the characters in the message. Suppose we prepare a function subprogram that accepts a character and returns the index of that character. This subprogram and the main program are given in Figure C-24. The function INDEX is called from line 6 and from line 11 of the main program. In each of the statements, the argument of the function is an array written with the subscript, which designates that a specific element of the respective array is to be used by the function subprogram. In the FUNCTION statement in the subprogram, a simple variable is used as the argument.

Also note the appearance of the STOP statement in the function subprogram in Figure C-24 in addition to the RETURN statement. All function subprograms must have at least one RETURN statement, although they may have more than one.

One obvious disadvantage of the program in Figure C-24 is that separate areas are reserved for array SYM in the main program and the subprogram. This can be avoided by the COMMON declaration, which we shall examine shortly.

As our final example of a function subprogram, we shall examine a problem in which arrays are used as arguments. Figure 4-3 presents the flowchart for a


```

      $JOB              1105.50003
1      DO 1 K=5780,5790,2
2      WRITE(6,2)K
3      2 FORMAT(' FACTORS OF ',I5)
4      NUMBER=K
5      J=2
6      3 IF(J.GT.NUMBER) GO TO 1
7      NUMBER=IFCT(NUMBER,J)
8      J=J+1
9      GO TO 3
10     1 CONTINUE
11     STOP
12     END

13     FUNCTION IFCT(NUMBER,J)
14     IFCT=NUMBER
15     2 K=IFCT/J
16     IF(K*J.NE.IFCT) RETURN
17     WRITE(6,1)J
18     1 FORMAT(1X,I5,' IS A FACTOR')
19     IFCT=K
20     GO TO 2
21     END

```

```

      $ENTRY
FACTORS OF 5780
      2 IS A FACTOR
      2 IS A FACTOR
      5 IS A FACTOR
      17 IS A FACTOR
      17 IS A FACTOR
FACTORS OF 5782
      2 IS A FACTOR
      7 IS A FACTOR
      7 IS A FACTOR
      59 IS A FACTOR
FACTORS OF 5784
      2 IS A FACTOR
      2 IS A FACTOR
      2 IS A FACTOR
      3 IS A FACTOR
      241 IS A FACTOR
FACTORS OF 5786
      2 IS A FACTOR
      11 IS A FACTOR
      263 IS A FACTOR
FACTORS OF 5788
      2 IS A FACTOR
      2 IS A FACTOR
      1447 IS A FACTOR
FACTORS OF 5790
      2 IS A FACTOR
      3 IS A FACTOR
      5 IS A FACTOR
      193 IS A FACTOR

```

Figure C-22 Use of function subprogram to compute factors

```

$JOB                                1105,50003

1      INTEGER FACTOR
2      DO 1 K=5780,5790,2
3      WRITE(6,2)K
4      2 FORMAT(' FACTORS OF',I5)
5      NUMBER=K
6      J=2
7      3 IF(J.GT.NUMBER) GO TO 1
8      NUMBER=FACTOR(NUMBER,J)
9      J=J+1
10     GO TO 3
11     1 CONTINUE
12     STOP
13     END

14     INTEGER FUNCTION FACTOR(NUMBER,J)
15     FACTOR=NUMBER
16     2 K=FACTOR/J
17     IF(K*J.NE.FACTOR) RETURN
18     WRITE(6,1) J
19     1 FORMAT(1X,I5,' IS A FACTOR')
20     FACTOR=K
21     GO TO 2
22     END

$ENTRY
FACTORS OF 5780
2 IS A FACTOR
2 IS A FACTOR
5 IS A FACTOR
17 IS A FACTOR
17 IS A FACTOR
FACTORS OF 5782
2 IS A FACTOR
7 IS A FACTOR
7 IS A FACTOR
59 IS A FACTOR
FACTORS OF 5784
2 IS A FACTOR
2 IS A FACTOR
2 IS A FACTOR
3 IS A FACTOR
241 IS A FACTOR
FACTORS OF 5786
2 IS A FACTOR
11 IS A FACTOR
263 IS A FACTOR
FACTORS OF 5788
2 IS A FACTOR
2 IS A FACTOR
1447 IS A FACTOR
FACTORS OF 5790
2 IS A FACTOR
3 IS A FACTOR
5 IS A FACTOR
193 IS A FACTOR

```

```

1  $JOB      1105,50003
2  INTEGER KEY(8),MSG(16),SYM(27),CRYP(16),NKEY(8)
3  DATA SYM/ 'A','B','C','D','E','F','G','H','I','J','K','L','M',
4  1 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'/
5  READ(5,1)(KEY(J),J=1,8),(MSG(J),J=1,16)
6  1 FORMAT(8A1/16A1)
7  DO 2 J=1,8
8  2 NKEY(J)=INDEX(KEY(J))
9  J=0
10 DO 3 L=1,16
11 J=J+1
12 IF(J.EQ.9) J=1
13 M=INDEX(MSG(L)) + NKEY(J) +10
14 A=M/27
15 M=M-27*INT(A)
16 3 CRYP(L)=SYM(M+1)
17 WRITE(6,4)(CRYP(J),J=1,16)
18 4 FORMAT('UCRYPTOGRAM',5X,16A1)
19 STOP
20 END
21
22 FUNCTION INDEX(I)
23 INTEGER SYM(27)
24 DATA SYM/ 'A','B','C','D','E','F','G','H','I','J','K','L','M',
25 1 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'/
26 DO 1 J=1,27
27 IF(SYM(J).EQ.I) GO TO 2
28 1 CONTINUE
29 WRITE(6,3)I
30 3 FORMAT('OINVALID CHARACTER ',A1)
31 STOP
32 2 INDEX=J-1
33 RETURN
34 END

```

```

$ENTRY
CRYPTOGRAM UCWGEGOBM XCDGPZ

```

a. Program with output

Figure C-24 Use of function subprogram in cryptogram program

COMPUTER
HE HAD A BAD DAY

b. Input data

Figure C-24 (Continued)

program that reads an initial inventory, updates that inventory for items received, and then prints the final inventory. Figure C-16 contains Fortran programs for this problem.

In this section, we shall expand the program to encompass updating of the inventory for items shipped as well as received. The resulting program in Figure C-25 uses function subprogram LOOKUP to determine the index of the entry in array STKNO to be updated. In the function calls in lines 6 and 11, array STOCK written without subscripts is used as an argument. This indicates that the entire array is to be available to the subprogram. In the FUNCTION statement, this array is again written without subscripts. In fact, array names written with subscripts are not accepted as arguments in FUNCTION statements.

Note that array STOCK and simple variable STKNO must be declared INTEGER in both the main program and in the subprogram, a consequence of the fact that the main program and the subprogram are compiled separately.

As a final note in this section, it is perfectly legitimate to change names between the arguments appearing in the main program and arguments appearing in the FUNCTION statement. The fact that we did this only in the cryptogram program in Figure C-24 was not meant to mislead the reader.

Subroutine

In Fortran, the subroutine is very similar to the function subprogram except that no value is associated with the name of the subroutine. The first statement in the subroutine is the SUBROUTINE statement. For procedure ORDER in Figure 5-11b, the first statement would be

```
SUBROUTINE ORDER (X)
```

Like the function subprogram the subroutine may also have more than one RETURN statement, but unlike the function subprogram it is not mandatory that the subroutine have a RETURN statement. Call by address is always used for subroutine arguments.

In the calling program, the subroutine is referenced by the CALL statement, a typical example being

```
CALL ORDER (A)
```

Figure C-26 gives the Fortran program corresponding to Figure 5-11 with the exception that NELE has not been declared global and therefore appears in the argument list.

In general, a function subprogram can be converted to a subroutine by simply adding an argument to the subroutine to accommodate the value associated with

```

$JOB      1105.50003

1      INTEGER STOCK(20),QUAN(20),STKNO
2      READ(5,1) ITEMS,(STOCK(J),QUAN(J),J=1,ITEMS)
3      1 FORMAT(I7/(I10,I8))
4      4 READ(5,2)STKNO,NREC
5      2 FORMAT(I10,I8)
6      IF(STKNO.EQ.0) GO TO 3
7      J=LOOKUP(STOCK,STKNO,ITEMS)
8      IF(J.NE.0) QUAN(J)=QUAN(J) + NREC
9      GO TO 4
10     3 READ(5,2)STKNO,NSHP
11     IF(STKNO.EQ.0) GO TO 5
12     J=LOOKUP(STOCK,STKNO,ITEMS)
13     IF(J.NE.0) QUAN(J)=QUAN(J)-NSHP
14     GO TO 3
15     5 WRITE(6,6)(STOCK(J),QUAN(J),J=1,ITEMS)
16     6 FORMAT(1X,I4,2X,I4)
17     STOP
18     END

19     FUNCTION LOOKUP(STOCK,STKNO,ITEMS)
20     INTEGER STOCK(20),STKNO
21     DO 1 LOOKUP=1,ITEMS
22     IF(STOCK(LOOKUP).EQ.STKNO) RETURN
23     1 CONTINUE
24     WRITE(6,2) STKNO
25     2 FORMAT(' NO ITEM NUMBER ',I5)
26     LOOKUP=0
27     RETURN
28     END

```

\$ENTRY

```

NO ITEM NUMBER  1985
NO ITEM NUMBER   356
NO ITEM NUMBER   964
NO ITEM NUMBER  1092
NO ITEM NUMBER  2435
NO ITEM NUMBER   632
1108    250
1068    245
1972    210
1066    200
1984    200

```

a. Program with output

Figure C-25 Use of function subprogram in inventory program

the function name. In the program in Figure C-27, variable J has been added to the argument list to convert LOOKUP from a function to a subroutine. This also necessitates the addition of CALL statements in the main program. Like function subprograms, the subroutine is compiled separately from the main program, which permits duplicate statement numbers, duplicate variable names, etc., but also necessitates repetition of declaration statements such as the INTEGER statement in Figure C-27.

5	
1108	150
1068	245
1972	110
1066	100
1984	200
1972	100
1108	100
1985	021
1066	100
0	31
356	527
964	037
1092	82
2435	578
632	1097
0	062

b. Input data

Figure C-25 (Continued)

```

$JOB 1105.50003
1      DIMENSION A(10),B(10)
2      READ(5,1)NELE,(A(J),J=1,NELE),(B(J),J=1,NELE)
3      1 FORMAT(15/(8F10.0))
4      CALL ORDER(A,NELE)
5      CALL ORDER(B,NELE)
6      WRITE(6,2)(A(J),B(J),J=1,NELE)
7      2 FORMAT(1X,2F10.3)
8      STOP
9      END

10     SUBROUTINE ORDER(X,NELE)
11     DIMENSION X(10)
12     N1=NELE-1
13     DO 1 J=1,N1
14     M=J
15     L=J+1
16     DO 2 K=L,NELE
17     2 IF(X(K).LT.X(M))M=K
18     T=X(J)
19     X(J)=X(M)
20     1 X(M)=T
21     RETURN
22     END

$ENTRY
96.000 34.000
175.000 176.000
181.000 290.000
35210.000 67981.000

```

Figure C-26 Example of a subroutine


```

$JOB      1105.50003
1      INTEGER STOCK(20),QUAN(20),STKNO
2      READ(5,1)ITEMS,(STOCK(J),QUAN(J),J=1,ITEMS)
3      1 FORMAT(17/(110,18))
4      4 READ(5,2)STKNO,NREC
5      2 FORMAT(110,18)
6      IF(STKNO.EQ.0) GO TO 3
7      CALL LOOKUP(STOCK,STKNO,ITEMS,J)
8      IF(J.NE.0)QUAN(J)=QUAN(J)+NREC
9      GO TO 4
10     3 READ(5,2)STKNO,NSHP
11     IF(STKNO.EQ.0) GO TO 5
12     CALL LOOKUP(STOCK,STKNO,ITEMS,J)
13     IF(J.NE.0)QUAN(J)=QUAN(J)-NSHP
14     GO TO 3
15     5 WRITE(6,6)(STOCK(J),QUAN(J),J=1,ITEMS)
16     6 FORMAT(1X,14,2X,14)
17     STOP
18     END

19     SUBROUTINE LOOKUP(STOCK,STKNO,ITEMS,J)
20     INTEGER STOCK(20),STKNO
21     DO 1 J=1,ITEMS
22     IF(STOCK(J).EQ.STKNO) RETURN
23     1 CONTINUE
24     WRITE(6,2)STKNO
25     2 FORMAT(' NO ITEM NUMBER ',15)
26     J=0
27     RETURN
28     END

```

\$ENTRY

```

NO ITEM NUMBER  1985
NO ITEM NUMBER   356
NO ITEM NUMBER   964
NO ITEM NUMBER  1092
NO ITEM NUMBER  2435
NO ITEM NUMBER   632
1108    250
1068    245
1972    210
1066    200
1984    200

```

a. Program with output

Figure C-27 Use of subroutine in inventory program

Although in the majority of the cases, the use of a function subprogram as opposed to a subroutine is primarily a matter of personal preference, the subroutine offers a clear advantage when the result is in the form of an array as opposed to a simple variable. As an example, suppose we modify the cryptogram program in Figure C-24. Instead of looking up the index of each character in the keyword and storing in a separate array, suppose we store the result in the array that originally contained the keyword. The same operation can be performed on the message itself and then the cryptogram formed.

5	
1108	150
1068	245
1972	110
1066	100
1984	200
1972	100
1108	100
1985	021
1066	100
0	31
356	527
964	037
1092	82
2435	578
632	1097
0	062

b. Input data
Figure C-27 (Continued)

The program in Figure C-28 uses subroutine CRYPTO to convert the characters in the array to their corresponding indexes according to Table 4-4. In this case, the result computed in the subprogram appears in the array in which the original data appeared, thereby making the subroutine more appropriate than the function subprogram. This approach still suffers from the disadvantage of the dual reservation of storage and declaration of array SYM in the main program and the subroutine.

COMMON Declaration

The statement for global declaration in Fortran is the COMMON statement. The COMMON block is established by the statement in the calling program and must reference the same number of storage locations as does the COMMON statement in the subprogram. For example, variable NELE is declared in common by the COMMON statements in Figure C-29.

Note that we only said that the respective COMMON statements must reference the same number of storage locations. For example, suppose the COMMON statement in the main program is

```
COMMON A,B,C
```

and in the subprogram is

```
COMMON B(3)
```

This statement reserves three locations for array B in the common area (B should not appear in another specification statement such as DIMENSION). But since these same three locations were defined as A, B, and C in the main program, then B(1) in the subprogram is synonymous with variable A in the main program, B(2) with B, and B(3) with C.

```

1  $JOB      1105.5003
2  INTEGER KEY(16),MSG(16),SYM(27),CRYP(16)
3  DATA SYM/' ','A','B','C','D','E','F','G','H','I','J','K','L','M',
4  1 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'/
5  READ(5,1)(KEY(J),J=1,8),(MSG(J),J=1,16)
6  1 FORMAT(8A1/16A1)
7  CALL CRYPTO(KEY,8)
8  CALL CRYPTO(MSG,16)
9  J=0
10 DO 3 L=1,16
11 J=J+1
12 IF(J.EQ.9)J=1
13 M=MSG(L) + KEY(J) +10
14 A=M/27
15 M=M-27*INT(A)
16 3 CRYP(L)=SYM(M+1)
17 WRITE(6,4)CRYP
18 4 FORMAT('UCRYPTOGRAM',5X,16A1)
19 STOP
20 END
21
22 SUBROUTINE CRYPTO(K,N)
23 INTEGER K(16),SYM(27)
24 DATA SYM/' ','A','B','C','D','E','F','G','H','I','J','K','L','M',
25 1 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'/
26 DO 2 I=1,N
27 DO 1 J=1,27
28 IF(SYM(J).EQ.K(I)) GO TO 2
29 1 CONTINUE
30 WRITE(6,3)K(I)
31 3 FORMAT('OINVALID CHARACTER ',A1)
32 STOP
33 2 K(I)=J-1
34 RETURN
35 END
36
37 $ENTRY
38 CRYPTOGRAM UCWGEGBM XCDGPZ
39 a. Program with output

```

Figure C-28 Use of subroutine in cryptogram program

COMPUTER
HE HAD A BAD DAY

b. Input data

Figure C-28 (Continued)

In some cases, the use of COMMON can remove all variables from the argument list. For example, all of the arguments for subroutine LOOKUP in Figure C-27 can be placed in COMMON to produce the program in Figure C-30. Whenever the same variable or array is used for a particular argument each time the subprogram is called, that variable or array can be declared in common and deleted from the argument list.

In the main program in Figure C-30, array STOCK is declared to contain twenty elements in the INTEGER statement. In the subroutine, STOCK is declared to contain twenty elements in the COMMON statement. The array declaration may be made in either statement but not simultaneously in both.

Although the COMMON declaration may be used with function subprograms,

```

      $JOB                      1105.50003
1      DIMENSION A(10),B(10)
2      COMMON NELE
3      READ(5,1)NELE,(A(J),J=1,NELE),(B(J),J=1,NELE)
4      CALL ORDER(A)
5      CALL ORDER(B)
6      WRITE(6,2)(A(J),B(J),J=1,NELE)
7      1 FORMAT(15/(8F10.0))
8      2 FORMAT(1X,2F10.3)
9      STOP
10     END

11     SUBROUTINE ORDER(X)
12     COMMON NELE
13     DIMENSION X(10)
14     N1=NELE-1
15     DO 1 J=1,N1
16     M=J
17     L=J+1
18     DO 2 K=L,NELE
19     2 IF(X(K).LT.X(M))M=K
20     T=X(J)
21     X(J)=X(M)
22     1 X(M)=T
23     RETURN
24     END

      $ENTRY
      96.000      34.000
      175.000     176.000
      181.000     290.000
      35210.000  67981.000

```

Figure C-29 Use of the COMMON statement

it may not be used to delete all of the arguments. That is, a function subprogram must have at least one argument.

As our final example of the use of the COMMON declaration, we shall remove the duplicate reservation of storage from the program in Figure C-24. A problem

```

$JOB          1105.50003
1      INTEGER STOCK(20),QUAN(20),STKNO
2      COMMON STOCK,STKNO,ITEMS,J
3      READ(5,1)ITEMS,(STOCK(J),QUAN(J),J=1,ITEMS)
4      1 FORMAT(I7/(I1),I8))
5      4 READ(5,2)STKNO,NREC
6      2 FORMAT(I10,I8)
7      IF(STKNO.EQ.0) GO TO 3
8      CALL LOOKUP
9      IF(J.NE.0) QUAN(J)=QUAN(J) + NREC
10     GO TO 4
11     3 READ(5,2)STKNO,NSHP
12     IF(STKNO.EQ.0) GO TO 5
13     CALL LOOKUP
14     IF(J.NE.0) QUAN(J)=QUAN(J)-NSHP
15     GO TO 3
16     5 WRITE(6,6)(STOCK(J),QUAN(J),J=1,ITEMS)
17     6 FORMAT(1X,I4,2X,I4)
18     STOP
19     END

20     SUBROUTINE LOOKUP
21     INTEGER STOCK,STKNO
22     COMMON STOCK(20),STKNO,ITEMS,J
23     DO 1 J=1,ITEMS
24     IF(STOCK(J).EQ.STKNO) RETURN
25     1 CONTINUE
26     WRITE(6,2)STKNO
27     2 FORMAT(' NO ITEM NUMBER ',I5)
28     J=0
29     RETURN
30     END

$ENTRY
NO ITEM NUMBER 1985
NO ITEM NUMBER 356
NO ITEM NUMBER 964
NO ITEM NUMBER 1092
NO ITEM NUMBER 2435
NO ITEM NUMBER 632
1108 250
1068 245
1972 210
1066 200
1984 200

```

a. Program with output

Figure C-30 Use of COMMON declaration in inventory program

520

5	
1108	150
1068	245
1972	110
1066	100
1984	200
1972	100
1108	100
1985	021
1066	100
0	31
356	527
964	037
1092	82
2435	578
632	1097
0	062

b. Input data
Figure C-30 (Continued)

arises from the fact that some systems do not permit the DATA statement to initialize variables or arrays that appear in COMMON, although a special feature known as the BLOCK DATA subprogram is provided for this purpose. However, we shall not introduce the BLOCK DATA feature. One way to circumvent this problem is to read the data into the array in COMMON, as illustrated by the program in Figure C-31.

C.14. ADDITIONAL FEATURES OF FORTRAN

This appendix has only covered a subset of the features of Fortran but hopefully a subset containing the most frequently used features. Additional features include:

- EQUIVALENCE statement (used to assign two or more variable names to the same storage location. The use of this statement in conjunction with the COMMON statement offers some interesting possibilities.)
- Adjustable dimensions
- Double precision features
- Features dealing with complex numbers
- Logical variable features
- NAMelist input/output features
- Direct access input/output features
- BLOCK DATA
- EXTERNAL declaration
- Labeled COMMON
- Multiple entry and return features
- Assigned GOTO


```

$JOB      1105.50003
1  INTEGER KEY(8),MSG(16),SYM,CRYP(16),NKEY(8)
2  COMMON SYM(27)
3  READ(5,1)(SYM(J),J=1,27),(KEY(J),J=1,8),(MSG(J),J=1,16)
4  1 FORMAT(27A1/8A1/16A1)
5  DO 2 J=1,8
6  2 NKEY(J)=INDEX(KEY(J))
7  J=0
8  DO 3 L=1,16
9  J=J+1
10 IF(J.EQ.9)J=1
11 M=INDEX(MSG(L)) + NKEY(J) + 10
12 A=M/27
13 M=M-27*INT(A)
14 3 CRYP(L)=SYM(M+1)
15 WRITE(6,4)(CRYP(J),J=1,16)
16 4 FORMAT('0CRYPTOGRAM ',5X,16A1)
17 STOP
18 END

19 FUNCTION INDEX(I)
20 INTEGER SYM
21 COMMON SYM(27)
22 DO 1 J=1,27
23 IF(SYM(J).EQ.I) GO TO 2
24 1 CONTINUE
25 WRITE(6,3)I
26 3 FORMAT('0INVALID CHARACTER ',A1)
27 STOP
28 2 INDEX=J-1
29 RETURN
30 END

```

```

$ENTRY
CRYPTOGRAM      UCWGEGBM XCDGPZ
a. Program with output
Figure C-31 Use of COMMON declaration in cryptogram program

```

```
      ABCDEFGHIJKLMNOPQRSTUVWXYZ  
COMPUTER  
HE HAC A BAD DAY
```

b. Input data

Figure C-31 (Continued)

Many specific implementations of Fortran contain even more features. Additional information on those features can be obtained from the numerous programming manuals dedicated to Fortran.



PL/I

PL/I is a multipurpose programming language; that is, it is designed to be used by both scientific and commercial programmers and to give these programmers a language powerful enough to handle all their problems and to make the complete power of the digital computer available to them.

For many years programming languages were thought of and designed for one particular class of user, for example, Fortran for scientific users and Cobol for commercial users. Now as computers themselves are used in a broader spectrum of applications, it is logical that the programming languages involved also become more flexible. PL/I was developed by IBM in conjunction with the scientific customer user's group SHARE and with the commercial customer user's group GUIDE.

PL/I is constructed so that the individual programmer can move and operate easily at his own level of experience—even if this level is low or high. The language has many automatic and default conditions to assist the beginning programmer, while the person with extensive experience may detail each and every step of his complicated program.

D-1. BASIC ELEMENTS

Character Set

The PL/I language is based on a set of sixty characters. These include the ten digits 0 – 9 and an extended alphabet containing twenty-nine characters—the

The material in this appendix in Sections D-2, D-3, part of D-6, D-7, and D-9 are reprinted by permission from "A PL/I Primer," Publication C28-6808-0. © 1965 by International Business Machine Corporation.

524 twenty-six letters of the English alphabet, the currency symbol (\$), the commercial “at” sign (@), and the number sign (#). In addition, it contains the following twenty-one special characters:

Name	Character
Blank	
Equals sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Point or period	.
Single quotation mark or apostrophe	'
Percent symbol	%
Semicolon	;
Colon	:
“Not” symbol	¬
“And” symbol	&
“Or” symbol	
“Greater than” symbol	>
“Less than” symbol	<
Break character	—
Question mark	?

These special characters also may be combined together to create other symbols. Some PL/I compilers are based upon a special 48-character set, but this will not be presented in detail in this appendix.

Program Comments

The pair of characters, /*, is used in PL/I to indicate the beginning of a programmer’s comment within a program. These same characters can be reversed, */, in order to indicate the end of a comment. When these characters are used in this manner, no blanks or other characters can separate the two characters. The comment that is contained between these character pairs may contain any combination of characters other than the */ combination which would be interpreted as the end of the comment. It is also possible that such comments contain characters that are not in the PL/I character set just as long as such characters are recognized by the computer hardware involved.

Identifiers

In PL/I a combination of alphameric characters and break characters may be used as identifiers which are the names of data items, files, and special condi-

tions, and as labels for statements. An identifier must begin with one of the twenty-nine alphabetic characters and it cannot contain blanks. PL/I does not allow identifiers to exceed thirty-one characters in length, though some compilers further restrict this allowable length. Some examples of identifiers are as follows:

```
X
HAPPL
ANSWER
ANS32
READ
#69
BIG_ANS
```

Note that the actual words of the PL/I language (such as READ, WRITE, IF, etc.) are also allowable identifiers. Such words are referred to as keywords, and in specific contexts they have a fixed meaning to the PL/I compiler, that is, they specify action to be taken, the nature of data, or the purpose of a name. A break character may be used to improve the readability of an identifier; this was illustrated in BIG_ANS. The use of the break character in this way is necessary since blanks are not permitted in identifiers.

Data

In appreciating data it is appropriate to list several different “levels” of data, that is, different representations of the same values. These are as follows:

1. Raw data: the values to be processed and the information characteristic to the problem to be solved.
2. The representation of values in the program.
3. Compiler input data: the representation as it is punched into a card or is entered from a typewriter terminal. Such data is translated into machine-language data.
4. Internal data: the representation as it is maintained inside the computer.

The use of data hereafter will refer to the representation of values as written in a PL/I program (level 2). Reference to a PL/I data item is made by using either a variable or a constant. A *variable* is a symbolic name (an identifier) having a value that changes (or that may change) during the execution of a program. The variable is not the data item itself—it is the name of the data item—and its value at any specific time is the data item to which it refers at that time.

A *constant* is unchanging—it is not given a symbolic name. Consider the following statement:

```
PAY = HOURS * 1.40;
```

In this statement PAY and HOURS are variables and 1.40 is a constant.

A constant does more than state a value because it demonstrates characteristics of the data item itself. For example 1.40 indicates that the data item is a decimal number of three digits and two of these digits are to the right of the decimal point. On the other hand, characteristics of variables are not immediately apparent in the name itself and these characteristics, which are called *attributes* in PL/I, must be known. Certain keywords are used to describe these attributes of a variable when it first appears in the program, and this will be presented later.

Expressions

An expression may be a single constant or a name or it may be a combination of them including operators and other delimiters. An arithmetic expression in PL/I combines arithmetic data, identifiers, and arithmetic operators. These arithmetic expressions may include

Addition	X + Y
Subtraction	X – Y
Multiplication	X * Y
Division	X/Y
Exponentiation	X**Y

It is also possible to combine a number of arithmetic operations into a single expression

$$((X + Y - Z)/R)**S$$

Parentheses are used to indicate that the parenthesized portion is treated as a single value in relation to its surrounding arithmetic operations. Innermost parentheses are evaluated first.

Although the expression may contain more than one data item, the expression represents the single value obtained after the evaluation of the expression.

Fixed-point and Floating-point Notation

Fixed-point notation is used in our everyday work. The 1.40 used earlier is an example of fixed-point notation, and it illustrates that fixed-point notation consists of digits and a single point that specifies the beginning of a fraction. In some cases this point is omitted if there is no fractional part.

Floating-point notation is often used in data processing for values that are much larger or much smaller than would be convenient to specify in fixed-point notation. In PL/I floating-point data is written as a field of digits followed by the letter E. This is followed by an optionally signed decimal exponent. The first field may contain a point and may be preceded by a + or – sign. Some examples of decimal floating-point data are as follows:

21E18
21E–18

21.34E18
2134E16

In either decimal or binary floating-point notation the purpose of the exponent is to indicate the movement of digit places which should be performed on the point in the first field. Point movement to the right is predicated by a positive exponent and to the left when a negative exponent is specified. For example, the first illustration given above actually represents the following number in decimal fixed-point notation

21000000000000000000.

This illustrates quite vividly that the sign of the exponent indicates the direction in which the point is moving, and it does not indicate whether the data item itself is positive or negative. If no sign (or a + sign) precedes the first field, the indication is that the data item is not less than zero.

Binary floating-point data is comparable to decimal floating-point data except that each digit position shift of a binary point multiplies or divides the number by 2.

D-2. DATA TYPES

There are three types of data commonly used in a PL/I program: arithmetic, string, and statement label.[†] *Arithmetic data* items are binary and decimal fixed-point or floating-point numbers. *String data* items are combinations of alphameric and special characters or combinations of binary digits. *Statement-label data* items are character strings used as label prefixes.

The following discussion covers each of these types of data, showing how data constants are written in a program and how data variables are defined and used.

Arithmetic Data

An item of arithmetic data is one with numeric value, that is, a number. It may be written in a program as a decimal or binary constant, or it may be the value of a variable.

Any number has certain characteristics. In the case of a constant, these characteristics are apparent. The notation in which a constant is written tells whether the data item has a decimal or binary *base* and whether it has a fixed-point or floating-point *scale*, and it states the value of the item. The number 6.5, used in an arithmetic expression, is a decimal, fixed-point number with the value of six and one-half. This number is expressed by using only two digits; it is not written 06.5 or 6.50. The computer need reserve storage space for

[†]Certain other types of variables, used in program control, are also classed as separate data types. They include task, pointer, event, and area variables.

only two digits. Although the decimal point is not stored in the computer, the result of an arithmetic computation is as if the point actually appeared; point alignment is maintained.

A data variable, since it is a name, does not demonstrate the characteristics of the data items it will represent. A reader would infer that the name WAGES would represent currency values. But a computer recognizes words only in the way it has been instructed to recognize them, or not at all; it assumes only what it has been instructed to assume.

Attributes must be declared for each variable so that the compiler can reserve sufficient storage space and use the proper internal notation for data items assigned to the variable. (Different kinds of data use different internal notation; for example, decimal notation and binary notation for the same arithmetic value are stored in different bit configurations.)

An arithmetic variable must have a DECIMAL or a BINARY attribute, a FIXED or a FLOAT attribute, and a *precision* attribute.

For example, the DECLARE statement for WAGES might be

```
DECLARE WAGES DECIMAL FIXED (5,2);
```

Note that blanks must always be used to separate the keyword DECLARE, the name, and the attributes that are keywords. The precision specification must be enclosed in parentheses and must immediately follow (with or without an intervening blank) the base attribute (DECIMAL or BINARY) or the scale attribute (FIXED or FLOAT).

The precision attribute specifies the *maximum* number of digits of any data item to be assigned to the variable, and it specifies the location of the assumed decimal point. In the above example the precision attribute (5,2) declared for WAGES specifies that no data item assigned to WAGES should contain more than five digits and that each data item is assumed to have a decimal point immediately preceding the last two digits; that is, each number will have two fractional digits (in this case, to express a value in cents).

Thus, precision implies a value *range*. For example, WAGES could represent any value from -999.99 to 999.99. If any number assigned to WAGES has fewer than three integer and two fractional digits, zeroes will be inserted to express the value as a five-digit number. If a number has too many digits, the excess will be lost—on the right if too many fractional digits are specified—on the left if too many integer digits are specified.

If TOTAL is declared to be a decimal fixed-point variable with precision of (10,2) any data item validly assigned to TOTAL, that is, any item representing numeric value, will be expressed as a decimal fixed-point number of ten digits, including two fractional positions. Assume X represents a decimal floating-point data item and Y represents a binary fixed-point data item. In execution of the statement $TOTAL = X + Y$, the data represented by X and Y would be converted to a common base and scale, the two numbers would be added, and the computed value would be converted to decimal fixed-point notation before assignment, that is, before it becomes the current value of TOTAL.

An arithmetic constant may be preceded by a plus sign or a minus sign. If unsigned, the constant is assumed to have a positive value. The sign of an arithmetic data item, unlike the point, is recorded in storage along with the data item. Consequently, the precision attribute (10,2) instructs the compiler to reserve sufficient storage for ten digits *plus the sign*.

Fixed-Point Decimal Data. A fixed-point decimal data item consists of one or more decimal digits. A decimal point may be included. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit.

Examples of fixed-point decimal constants as written in a program are

```
3.141593
-5280
455.3
.00003
234.
```

The keywords for decimal fixed-point variables are `DECIMAL` and `FIXED`. Precision is stated by means of two decimal integers, separated by a comma and enclosed in parentheses. If only one number is specified, the items are assumed to be integers.

To define `PI` (3.141593) in this way, the following statement could be used:

```
DECLARE PI FIXED DECIMAL (7,6);
```

This defines the identifier `PI` as a fixed-point decimal item of not more than seven digits, six of which are to the right of the decimal point. This declaration, of course, just authorizes use of the identifier `PI` in the program. No value has been assigned. This could be done later with an assignment statement, such as

```
PI = 3.141593;
```

The value could also be assigned in the `DECLARE` statement, specifying the `INITIAL` attribute, as follows:

```
DECLARE PI FIXED DECIMAL (7,6) INITIAL (3.141593);
```

This not only defines the identifier `PI` but gives it an *initial* value of 3.141593. The value may be retained throughout the program, as is probable in this case, or it may be changed during execution by an assignment statement.

The statement of precision is critical in the definition of a variable. A programmer must know the size of the largest number that will be assigned to the variable during execution of the program and define the variable accordingly. Consider this statement:

```
SUM = X + Y;
```

If the value of `X` is 456.3 and the value of `Y` is .387, the result of this calculation

530 would be 456.687. The following list shows how this result would be assigned to SUM under several possible precision conditions:

Variable declaration	Assigned value
SUM FIXED DECIMAL (6,3)	456.687
SUM FIXED DECIMAL (8,4)	0456.6870
SUM FIXED DECIMAL (5,3)	56.687
SUM FIXED DECIMAL (6,2)	0456.68
SUM FIXED DECIMAL (6,4)	56.6870
SUM FIXED DECIMAL (6)	000456.

Note that alignment is on the decimal point, and zeroes are added or digits are deleted to make the item fit the precision specification. While a programmer must guard against unwanted truncation, he can also use this characteristic as a means of dropping undesired fractional digits at the time an assignment is made.

The maximum number of digits allowed in a decimal fixed-point data item is defined for each PL/I compiler. If precision is not specified when a variable is declared, the number of digits assumed depends upon the compiler in use. For example, the maximum might be fifteen with the assumed number—*default precision*—five. With default precision, the assumed location of the decimal point is immediately to the right of the rightmost digit.

Binary Fixed-Point Data. A binary fixed-point data item expresses an arithmetic value using binary notation. It is written as one or more binary digits with an optional binary point, followed by the letter B.

Examples of binary fixed-point constants as written in a program follow:

Constant	Decimal equivalent
10110B	22
11111B	31
-101B	-5
111.01B	7 1/4
-1011.111B	-11 7/8

A variable is declared to represent binary fixed-point data by specifying the BINARY, FIXED, and precision attributes.

Precision of a binary fixed-point variable is specified by two decimal integers, separated by a comma and enclosed in parentheses, to represent the maximum number of binary digits and the number of digits to the right of the binary point.

Following is an example of declaration for a binary fixed-point variable:

```
DECLARE FACTOR BINARY FIXED (20,2);
```

FACTOR is declared to be a variable that can represent an arithmetic data item as large as twenty binary digits, in addition to a plus or minus sign. The

decimal equivalent of that value range is from $-262,144.25$ through $+262,143.25^\dagger$

The maximum precision and default precision for binary fixed-point data are specified separately for each different PL/I compiler.

Decimal Floating-Point Data. A decimal floating-point data item is designated in a program by a field of decimal digits followed by the letter E followed by a decimal exponent that may be signed. The first field of digits may contain a decimal point and may be preceded by a plus or minus sign.

Examples of decimal floating-point data items are

```
15E23
15E-23
-4835E48
3141593E-6
.003141593E3
```

The last two examples represent an identical value.

The attributes that describe decimal floating-point variables are DECIMAL and FLOAT. Precision is stated with a decimal integer enclosed in parentheses. It specifies the number of digits to be maintained preceding the E. With most computers, floating-point data items are stored internally with the point immediately to the left of the first nonzero digit. If an item is assigned to a variable with a declared precision that is smaller than the field width of the assigned item, truncation will occur on the right. The least significant digit is the first that is lost.

Following is an example of declaration of a decimal floating-point variable:

```
DECLARE LIGHT_YEARS DECIMAL FLOAT (5);
```

If the data item .814235E14 were assigned to LIGHT_YEARS, the rightmost digit of the fraction would be lost. The item still would represent a value of more than 81 trillion, but the single-digit truncation would decrease the value of the data item by 500 million. If the data item were .814E14, two zeroes would be inserted to maintain the declared precision, and it would be as if the item assigned were .81400E14.

Binary Floating-Point Data. A binary floating-point data item, when written in a program, consists of a field of binary digits followed by the letter E, followed by a decimal integer exponent followed by the letter B. The field of binary digits may contain a binary point and, of course, a plus or minus sign. The exponent may be signed. As with decimal data, the exponent indicates displacement of the binary point.

Examples of binary floating-point data are as follows:

[†]The value range is as it would be in an IBM System/360 computer. In some computers, the lowest value that can be represented by eighteen binary integer digits might be $-262,143$.

```
101101E5B
101.101E2B
11101E-28B
```

The attribute keywords for binary floating-point data are `BINARY` and `FLOAT`. Precision is specified by stating the number of digits preceding the `E`, with precision expressed as a decimal integer:

```
DECLARE X BINARY FLOAT (16);
```

With most computers, binary floating-point data are represented internally with the assumed binary point immediately preceding the first 1 digit.

String Data

A string is a connected sequence of characters (or binary digits) that is treated as a single data item. The length of the string is the number of characters (or binary digits) it contains.

There are two types of strings: character strings and bit strings.

Character-String Data. A character string can include any digit, letter, or special character recognized as a character by the user's computer system. Any blank included in a character string is considered an integral character of the data item and is included in the count of the length. Comments cannot be inserted within a character string.

Character-string constants, when written in a program, must be enclosed in single quotation marks. If an apostrophe or a single quotation mark is a character in a string, it must be written as two single quotation marks with no intervening blank.

The following are examples of character-string constants:

```
'LOGARITHM TABLE'
'PAGE 5'
'SHAKESPEARE'S ""HAMLET""'
'23842'
(2) 'WALLA '
```

In the last example, the parenthesized number indicates repetition of the characters and specifies the character string `'WALLA WALLA '` (the blank is included as one of the characters to be repeated). The repetition factor must be an unsigned decimal integer enclosed in parentheses to indicate the number of times the characters are to appear in the actual character string.

Although a character string may be entirely numeric, such a string cannot be used efficiently in arithmetic operations with computers in which the internal representation of numeric characters is different from the internal representation of arithmetic data.

Character-string data are declared to have the `CHARACTER` and length

attributes. Length is expressed by a decimal integer, enclosed in parentheses, which specifies the number of characters in the string. For example:

```
DECLARE NAME CHARACTER (15);
```

As with other data items declared in this way, the value of the variable, NAME, is to be assigned during execution of the program. Most data items, however, can also be given an initial value by declaring the name with the INITIAL attribute and listing the initial value. For example:

```
DECLARE NAME CHARACTER (15) INITIAL  
('JOHN DOE');
```

Although the declared length is 15, the length of the string assigned by the INITIAL attribute contains only eight characters. Blanks are added automatically on the right to fill out the length. The first character assigned is always left adjusted, with padding supplied on the right. In this case, the string would be stored as the characters JOHN DOE, followed by seven blanks.

A character string is assigned from left to right. If the actual string is longer than the declared length, the excess characters are truncated on the right.

Bit-String Data. A bit-string data item is written in a program as a series of binary digits enclosed in single quotation marks and followed by the letter B.

Bit strings are valuable for general use as logical switches that can be set to 1 or 0 as indicators that may be necessary later in the program for decision making.

Bit strings are increasingly used in information retrieval. Many “yes” or “no” answers can be recorded as a bit string in a relatively small area. For example, computer assistance in medical diagnosis makes wide use of bit strings.

Examples of bit-string constants are

```
'1'B  
'11111010110001'B  
(64) '0'B
```

The parenthesized number preceding the last example is a repetition factor which specifies that the following digit (or digits) is to be repeated the indicated number of times. The example shown would result in a string of sixty-four binary zeroes.

A bit-string variable is declared to have the BIT and length attributes. Length represents the number of binary digits in the string and is indicated by a decimal integer enclosed in parentheses. The letter B is not an actual part of the string and is not considered in the length specification.

Following is an example of declaration of a bit-string variable:

```
DECLARE SYMPTOMS BIT (64);
```

534 Like character strings, bit strings are assigned to variables from left to right. If a string is longer than the length declared for the variable, the rightmost digits are truncated; if shorter, padding, on the right, is with zeros.

Label Data

A statement label is an identifier written as a prefix to a statement so that, during execution, program control can be transferred to that statement through a reference to its label. A colon separates the label from the statement:

```
ABCDE: DISTANCE = RATE * TIME;
```

In the above example, ABCDE is the statement label. The statement can be executed either by normal sequential execution of instructions or by transferring control to this statement from some other point in the program by means of a statement such as GO TO ABCDE.

As used above, ABCDE can be classified further as a *statement-label constant*. A *statement-label variable* is an identifier that *refers* to statement-label constants. Consider the following example:

```
LBL_A:  statement;
      .
      .
      .
LBL_B:  statement;
      .
      .
      .
      LBL_X=LBL_A;
      .
      .
      .
      GO TO  LBL_X;
      .
      .
      .
```

LBL_A and LBL_B are statement-label constants because they are prefixed to statements. LBL_X is a statement-label variable. By assigning LBL_A to LBL_X, the statement GO TO LBL_X causes a transfer to the LBL_A statement. Elsewhere, the program may contain a statement assigning LBL_B to LBL_X. Then, any reference to LBL_X would be the same as a reference to LBL_B. This “value” of LBL_X is retained until another value is assigned to it.

A statement-label variable must be declared with the LABEL attribute, as follows:

```
DECLARE LBL_X LABEL;
```

D-3. EXPRESSIONS AND OPERATIONS

As discussed previously, any identifier other than a keyword that appears in a PL/I statement is an expression. A single variable is an expression, as is a single constant. Expressions can refer to arrays (array expressions); they can refer to structures (structure expressions); or they can refer to single items of data (called *scalar expressions*).

When one or more single expressions appear in conjunction with operators, the combination is an operational expression, and its type depends upon the nature of the operator. For example, an arithmetic expression is one that involves arithmetic operators.

An operator that precedes the variable or constant (the operand) is a *prefix operator* (as with $-A$ or $+A$); an operator that appears between operands is an *infix operator* (as in $A + B$ or $A - B$).

Arithmetic Operations

Most of the examples of operational expressions discussed so far in this book have been arithmetic expressions. The arithmetic operators, as noted before, are

+ - * / **

Of these, only the plus and minus sign can appear as prefix operators.

If the operands of an arithmetic expression differ in base or scale, they are converted to a common base and scale before evaluation is made. Since the result conforms with the attributes of the variable to which assignment is made, there is no need to discuss here the manner in which different operands are converted. A detailed discussion is included in the publication *PL/I: Language Specifications*.

Data operated upon in an arithmetic expression must have an arithmetic value.

Comparison Operations

There are three kinds of comparison operations: arithmetic, character, and bit.

The comparison operators are

< <= = ≠ >= >

None of the comparison operators can be a prefix operator.

Arithmetic comparison involves the comparison of signed arithmetic values.

Character comparison involves left-to-right, pair-by-pair comparison according to collating sequence. For example $A > B$, $B > C$. If the operands are of different lengths, the shorter is extended on the right with blanks.

Bit comparison involves left-to-right comparison of binary digits. If the operands are of different lengths, the shorter is extended on the right with binary zeros.

536 Concatenation Operations

Concatenation operations involve the chaining of characters, with no intervening blanks. The concatenation operator is ||, which is written as two “or” (|) symbols.

If the operands are bit-string, the result is a bit string. In all other cases, the result is a character string. If the operands are not bits or characters, they are converted to characters.

Examples:

If A is 7345 with an implied decimal point between the second and third digits, and if B is 8923 with an implied point before the first digit, A||B would result in the character string ‘73.45.8923’.

If C is ‘0111011’B and D is ‘111’B, the result of C||D is ‘0111011111’B.

If E is ‘ABC’ and F is ‘DEF’, the result of E||F is ‘ABCDEF’.

Bit-String Operations

Bit-string operations involve the following Boolean logical operators:

- ¬ not
- & and
- | or

The “not” sign always is a prefix operator; the “and” and “or” signs always are infix operators.

Bit-string operations are performed on a bit-by-bit basis, from left to right. If operands are not bit strings, they are converted before the operation is performed; if the operands are of different bit-string length, the shorter will be extended on the right with binary zeros.

The following table shows the result of a bit-to-bit comparison under each possible circumstance:

A	B	Not A	Not B	A and B	A or B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Consider the following examples:

A = ‘1101’B; B = ‘1111’B; C = ‘010’B;

¬A = ‘0010’B;

A|C = ‘1101’B;

A&B = ‘1101’B;

A&C = ‘0100’B;

¬A&B = ‘0010’B;

Order of Evaluation

Arithmetic expressions are evaluated according to the priority of the operator. Any expression enclosed in parentheses is evaluated before any other part of the expression.

Exponentiation (**), prefix +, and prefix − have the highest priority. These operations will be completed first, and if more than one of these operators appears in the same expression, they are evaluated from right to left.

Multiplication (*) and division (/) have the second priority. They are evaluated from left to right.

Addition (+) and subtraction (−) have the lowest priority. They are evaluated from left to right.

If any other order is desired, parentheses must be used to indicate the order. For example:

$A * B / C ** D$	is not equivalent to
$(A * B / C) ** D$	
$A * B / C$	is equivalent to
$(A * B) / C$	
$A / B + C$	is not equivalent to
$A / (B + C)$	

Bit-string operations, like arithmetic operations, are evaluated according to the priority of the operator. The “not” sign ($\neg$) has the highest priority; the “or” sign ($\mid$) has the lowest priority. Any expression enclosed in parentheses is evaluated first. For example:

$B \mid C \& D$	is equivalent to $B \mid (C \& D)$
$\neg B \mid C$	is not equivalent to $\neg (B \mid C)$

D-4. INPUT-OUTPUT OF STREAM-ORIENTED DATA

In PL/I all input and output operations are divided into two classes which are mutually exclusive of one another. The distinction between these two classes is made depending upon the nature of the *data-set*, that is, the data external to the program. In one case the data-set is regarded as a continuous stream of characters; this is referred to as *stream-oriented data*. On the other hand, if the data-set is regarded as being made up of discrete records, it is handled with *record-oriented* statements.[†] In actual practice these record-oriented statements are used easily only with data recorded in machine-coded form. This implies data which is in the computer or in temporary storage on magnetic tapes or discs, and it generally implies that the data stored in this fashion is not directly available for human use. In general these two classes of data are handled in PL/I via statements containing the following characteristics:

[†] Some subsets of PL/I provide only for stream-oriented statements.

	Stream-oriented	Record-oriented
Basic input statement keyword	GET	READ
Basic output statement keyword	PUT	WRITE
Data conversion performed	automatic	none
Data-set form	character	anything

Stream-oriented data transmission uses three different input-output modes or methods. These three different methods involve three different kinds of GET and PUT statements which range from those that require the programmer to provide no specifications (with the form and sequence of the data being standard and rigid) up to those which give the programmer complete control over the sequence, format, and editing of his data. The programmer can use any one of these three elementary methods without even knowing of the existence of the others.

In PL/I the GET and PUT statements handle the entire conversion of data between its external and its internal forms, that is, between the way the data is represented by the programmer and the way the data is represented within the machine's memory. Actually the statements GET and PUT convert the data and move the data between the working storage area within the computer's memory and the computer's input-output area. The computer's operating system handles the transmission between this area and the external recording device. The GET and PUT statements have no way of directly indicating which external storage device is to be used, but the GET and PUT statements may designate a FILE by name. The file name may be one which is assigned by the computer's operating system or it may be assigned by a control card at the time of the object program's execution. If no file is specified, the standard input and output files will be automatically selected, and they are named SYSIN and SYSPRINT.

The three basic input-output modes, if listed in the order of the simplest to the most complex, are[†]

- Data-directed
- List-directed
- Edit-directed

In addition to the GET and PUT statements involved in input-output operations, there are available some special purpose auxiliary statements for input and output. Some of them do not actually handle information transmission but they do play a role in facilitating this transmission. Several examples of these are DISPLAY, READ, WRITE, OPEN, and CLOSE.

It might be desirable at this point also to point out that there are special provisions attendant to the data list for output material which goes to a

[†]It was also pointed out that the data-directed mode is not included in some subsets of PL/I.

computer printer under the control of the computer's standard operating system. PL/I has *several carriage control format items*, and three of these—PAGE, LINE, and SKIP—apply both to list-directed and edit-directed output. These carriage control items dictate the manner in which the paper in the output printer is moved vertically before the items in the data list are printed.

Under standard output conditions the items on a data list are written on the output printer at standard tab positions. These tab positions are system independent, but for the IBM System/360, they are positions 10, 34, 58, 82, and 108.

Data-Directed I/O

Since data-directed I/O is the simplest mode available in PL/I, it follows as a result that forms of the data in this mode are most rigidly established. In both input and output of data-directed material the data on the external media appear as a series of simple assignment statements without semicolons and separated by blanks. Each series is handled as a bunch—we call it a “list”—and it is ended with a semicolon. Thus, the general form of the data on the external media is as follows:

variable = constant variable = constant . . . ;

Some specific examples of this general form might be as follows:

A = 23 B = 18 C = 2.1;
A = 21 B = 24 C = 4.2;

or

GROSS = 2073 PCT = 3.1;
GROSS = 1892 PCT = 2.6;
GROSS = 3084 PCT = 4.6;

or

HOPE = 18.432 TITLE = 'WORK';

On input of data-directed data there may be commas or blanks between “assignments” and the constants may have the + sign indicated or not. On output only blanks are used to separate between items in the data list, and the output prints a sign only if the constant is negative. The constant may be an arithmetic or a string constant, and to represent a complex constant an expression is necessary.

The simplest way to get data into the computer is the data-directed input mode using the statement

GET DATA;

This statement reads in one list of data up to the point of the semicolon, and it assigns the values given to the variables named in the data list. The statement names no FILE, and the input would be assumed by the compiler to be from the standard input file of the system—SYSIN.

It is possible to name an input file with data-directed input. In such a case the statement would take the following form:

GET FILE (TAPE2) DATA;

This type of statement calls for whatever input unit that has been designated as TAPE2 to be the input unit. Generally speaking this type of file designation is not normally used with data-directed input, and in the following input-output description the examples presented will be shown without this FILE option—but this option can be applied with any of them. When this option is used the external device that will correspond to the file name is established by the computer's operating system or by control cards loaded at object program execution time. Generally speaking, however, when external files are designated in this manner, the programmer normally finds it more advantageous to go to either list-directed or edit-directed data modes. For this reason this file option will not be discussed in detail.

There is a second way to get input data in the data-directed input mode. This is with a statement of the following type:

GET DATA (data-list);

or for example:

GET DATA (GROSS, PCT);

This type of statement functions exactly as did the first type, but the only difference is that the variables to be read are specified in the GET statement. In this type of statement the data list is a list of variables separated by commas, and this variable list need not be in the same order as the items in the data itself. It is also possible that there may be variables in the data list which do not appear in the data itself. We might have the following situation where the data string on the external media is as follows:

PCT = 3.1 GROSS = 2073;

and it will be read equally well by:

GET DATA (GROSS, PCT);

or by:

GET DATA (GROSS, PCT, AMT, SALES);

of course only the variables whose names appear on the external data string will have their values altered by the execution of such a GET statement as shown in the example above.

It is also possible that the external data string may include variables which are not named in the data statement. For example the following data string:

A = 10 B = 12 C = 18 D = 2.1;

can be read in with the statement:

GET DATA (A,C,D);

but only the internal stored values of A, C, and D will be changed by the execution of such a GET statement.

Practically speaking, for the output of data-directed information there is no output statement corresponding to the simple GET DATA statement, and the minimum data-directed output statement that is available is as follows:

PUT DATA (data list);

and this causes the output of an external string consisting of the name of each variable in order, followed by an equals sign and the current value of the variable. All separations are by blanks and the last item in the record is followed by a semicolon. As you might expect, signs are only shown for negative values. Since no FILE is named in the PUT statement, the output will be on the SYSPRINT which is the standard system output file. A simple PUT DATA statement can be used, but it will cause all variables known in the block to be printed.

The data-directed mode makes provision for the handling of individual data items, arrays, data structures, repetitive specifications, etc.

Data-directed output is suitable for subsequent use as data-directed input in almost all cases. There are some special cases, however, where data-directed output is not suitable, but these are more matters of curiosity than of concern, and they do not arise in ordinary circumstances.

Whenever character strings are output via the data-directed mode, they are written out enclosed in single quotes.

The number of digits which are output for numbers in the data-directed mode is equal to the precision of the internal data. If possible the binary or decimal point is adjusted to avoid using the E notation, and as a result, decimal points will not necessarily be aligned. If the value of the output entity cannot be expressed to the precision called for without using the E notation, however, the E notation will be used.

In summary it is easily seen that the data-directed output mode is useful in learning PL/I, in producing a very few named results, or in producing a program which has easily readable output for debugging or program testing. In general, however, it is not the mode which most experienced programmers like to use extensively.

List-Directed I/O

The list-directed input-output mode differs from the data-directed mode in that while the programmer specifies only the sequence and the names of the data to be input and/or output, the form of the external data is less rigid and the variable name does not appear with any item. In both input and output the data on the external medium is a sequence of constants, and each such series of constants is handled as a bunch (again, we call it a "list"). On input there may be commas or blanks between the individual items, but on output the separator

between items is the blank. As usual, signed constants may have the + sign indicated or not on input, but on output only the – sign is indicated. The constants in the data string may be arithmetic, string, or complex constants. In general, input data takes the following form (with blanks):

constant constant constant constant
constant . . .

or (with commas):

constant, constant, constant, constant,
constant, . . .

and as an example:

18, 21.7, 17.3, 19,
21.6, . . .

With the data in this prescribed form the input is obtained with the statement

GET LIST (data list);

The data will be read in as one list up to the number of variables named, and the values will be assigned to the variables named in the data list. Since the statement has named no FILE, the input is from the standard input file of the system—SYSIN—but a FILE option may be used to designate some other data source.

It is possible to skip a variable in the data list by putting a *null field* in the corresponding input item. This null field is created by omitting an item between two item separators; of course these separators must be commas.

The transmission of each list of constants with the GET LIST statement ends when the list is exhausted. The variables listed in the data list of the GET statement may be individual data items, arrays, structure names, or repetitive specifications involving any of these. The usual conversions are performed in the situation in which the form of the external data does not agree with that of the specified variable in the data list. In all cases the declared form of the variable will be controlling.

The output statement for list-directed output is as follows:

PUT LIST (data list);

and it functions to output a stream of external data items consisting of the current values of the variables in the data list. The order of the data list will be the output order, and the separation between items is by blanks. Signs are shown only for negative values. The data list for the PUT LIST statement may include the same type of data items that were permitted in the GET LIST data list—constant, array and structure expressions, and repetition specifications involving them.

List-directed output from one program is suitable for list-directed input to another program in most instances. Numeric data, whether carried internally in the machine in decimal or binary form or in fixed or floating format, will be in decimal form on output with the decimal point properly placed. The E format is resorted to only if the stated precision will not permit the expression of the number without use of the E format. Bit-string data is put out in single quotes with the identifying B.

In order to make titles or headings, character string data can appear directly in the data list in its literal form with quotes. It should be noted that character-string data so put out will be produced with the quotes having been stripped off. As a result, character-string data which is output in this way is unsuitable for later direct use as input.

Edit-Directed I/O

This mode is the most general and the most flexible I/O mode available to the PL/I programmer. This type of data transmission allows the programmer to specify the format by which data in the stream will be read and by which data items will be written. The basic format of the GET and PUT statements for edit-directed transmission is as follows:

GET FILE (file name) EDIT (data list) (format list);

and

PUT FILE (file name) EDIT (data list) (format list);

As in data-directed I/O and list-directed I/O, if the file specification is omitted, the standard system file is assumed. The data specification consists of two parts—the data list and the format list. Each must be enclosed within parentheses. The data list is the same as that described for list-directed transmission, i.e., a list of variables in a GET statement and a list of variables, constants, or other expressions in a PUT statement. The format list is a list of format descriptions, each of which describes the format of an individual data item. As you might expect, the first format item in the format list describes the format of the first data item in the data list, the second format item describes the second data item, etc. It is possible to have separate statements for the input-output operations and for the format list, but the usual practice is to follow the kind of statements illustrated.

In illustrating edit-directed input and output the practice of not utilizing the FILE option will be continued. As a result the input statement would appear as follows:

GET EDIT (data list) (format list);

and as an example:

GET EDIT (X,Y,Z) (F(10,2), F(12,4), F (8,0));

It is seen that there is no punctuation between the data list and the format list, but the individual format items themselves and the individual data items in the data list are separated by commas. Also note that the numeric codes following the format letter codes are themselves enclosed in parentheses and separated internally by commas.

An alternate way of accomplishing this identically same input operation is to put the format list in a separate labeled **FORMAT** statement and then to refer to this statement in the **GET** statement. This might be as follows:

```
GET EDIT (data list) (R (format statement label));
format statement label: FORMAT (format list);
```

and as an example:

```
GET EDIT (N,Y,Z) (R (FOR_1));
FOR_1: FORMAT (F(10,2), F(12,4), F(8,0));
```

This **R** in the **GET** statement format list simply implies that the format list will be **REMOTE** in the **FORMAT** statement having the label which follows the **R**. It is not necessary that this **FORMAT** statement follow the **GET** statement directly, but it may be anywhere in the program so long as its label is known to the **GET** statement.

Output operations for the edit-directed mode are just like those explained for input except that **PUT** is used instead of **GET**. Edit-directed output which is to appear on an output printer should employ carriage control format items.

No matter whether the format list is included in the input-output statement or whether it is given in a separate **FORMAT** statement, the execution of the specification is in reality the pairing of the items in the data list with the items in the format list, in order from left to right. It also involves the conversion of the data items according to the paired format item. If there are more format items than data items, the excess on the right is ignored. If there are more data items than there are format items, the format list is re-used.

The format list is made up of letter-number codes that describe the format of the data on the external medium, and some examples of this have been seen in earlier statements. This format item specifies how the data will appear on the external medium without any reference as to how it was or will be used internally, for example, the **E** format may be used with any internal data item. Any conversion which is needed will take place automatically.

There are six format item codes for use in **PL/I**:

- Fixed-point (**F**)
- Floating-point (**E**)
- Complex (**C**)
- Picture (**P**)
- Character-string (**A**)
- Bit-string (**B**)

Format Item Codes

The first format item code to be considered is the *fixed-point format* for numeric data only. The general form of this format code is:

$$F(w,d,p)$$

In this general form the letter F defines the fixed-point nature of the item, and the lower case italic letters *w*, *d*, and *p* stand for decimal values. These decimal values are not necessarily restricted to constants but they also may be variables or expressions. In general the programmer will find that most commonly these values are constants, but the possibility of their being variables or expressions allows the programmer to control field widths, decimal locations, or scale factors during the execution of the program; these expressions are evaluated and converted to integers when the format item is used. The external form of the number governed by the format will be decimal. If the field width is such that significant digits or the sign is lost, an error results.

The lower case italic *w* specifies the *width* of the data item's *field*, that is, the *total* number of digits or spaces involved, the letter *d* gives the number of *digits* to the *right* of the decimal place, and *p* specifies a *scale factor* which is 10 raised to the power of *p*. Normally the value for *p* is omitted, and in such cases it is considered to be 0. It is also possible to omit the values for both *d* and *p*, and in such cases both *d* and *p* will be considered to be 0. Of the three letters given, only *p* may be negative.

Note that these designators of width, decimal location, and scale factor (*w*, *d*, *p*) are enclosed in parentheses and are separated by commas. This is normal for all format items. In interpreting these items for input and output, however, it is best to consider them separately because there are some differences.

For input of information the decimal fixed-point number may be anywhere in the field of width *w*. In many cases there will already be a decimal point in the number, and this given decimal point will override any decimal point classification given by a specific value of *d*. In some cases no decimal point is given in the number itself, and in these cases it is always assumed to be *d* digits from the rightmost digit in the number given.

If the scale factor *p* is given, the input number is treated as though the decimal point actually appeared *p* places to the right for positive values of *p* or *p* places to the left for negative values of *p*. This is equivalent in numerical data to multiplying the value by 10 raised to the value of *p* after placing the decimal point according to *d* and then converting the value to the internal representation. The form of the internal representation of the data item is not set by the FORMAT type, but instead by the type of the corresponding value in the data list.

Some examples of input edit-directed fixed-point format items follow. For the format statement

(F(6), F(6,2), F(6,2,-4), F(6,2))

546 the following data stream ‘

51234698765432145612.345

would be read and stored as

512346.

9876.54

.321456

12.345 (A decimal existing in the input stream overrides the format specification, and it also accounts for a position in the width specification.)

During output the internal representation is put out as external data in the form of a decimal fixed-point number which is right-adjusted in a field of w ; blanks are inserted on the left to fill out the field. If only a value for w is given, the integer portion of the number is written out, no decimal point is shown, and any fractional value of the number is truncated. When values for both w and d are given, the number is written out with its decimal point shown, but only with d digits of the decimal fraction. If there are fractional values in excess of d , they are truncated. Again the number is right-adjusted in the field of w ; trailing zeroes are supplied when the number of fractional digits is less than d . When values for w , d , and p are all given, the internal data value is multiplied by 10 raised to the p power before conversion to the external form.

It is desirable to consider several examples to pinpoint some specific facts in these specifications. Note that $F(5)$ and $F(5,0)$ are different. The first format specification will yield an integer in a 5-space field and without a decimal point. The second specification, however, will produce the same integer in a 5-space field but with a decimal point printed, and as a result, the decimal point will cause all of the numbers in the field to be one space further over to the left. The output will be the same only if the number is ≤ 9999 .

It is also possible that one specification, such as d , may be omitted, for example, $F(5,,8)$. In this latter specification only the integer portion of the number without the decimal point will appear.

Negative numbers will have their minus sign in their external form, and a positive number will have no sign. The value of w should always be large enough to handle the digits needed with the decimal point and with the minus sign if negative numbers are possible.

For an example of output fixed-point format items, consider these data items:

- (a) 66.67175
- (b) -200.98
- (c) 555.

Outputted by the indicated formats, these data would appear as follows:

Format item	a	b	c
F(5)	66	-200	555
F(8,5)	66.67175	00.98000	55.00000
F(8,2)	66.67	-200.98	555.00
F(6,1,2)	6667.2	0098.0	5500.0

Floating-point format codes apply only to numeric data, and their general form is as follows:

$$E\ (w,d,s)$$

The letter E defines the floating-point nature of the data item, and the *w*, *d*, and *s* are decimal values. These decimal values may be constants, variables, or expressions, and the external form of the number described by the format *must* be decimal.

As might be suspected, the value of *w* gives the width of the field for the data item, the value of *d* gives the number of digits to the right of the decimal point in the mantissa, and *s* gives the total number of digits, not counting sign or decimal points, in the mantissa. The *s* may be omitted but values for *w* and *d* *must* be given. The values of *w* and *s* must be positive, but the value of *d* may be zero or positive. When *s* is specified, it must be less than seventeen digits; when *s* is not specified, *d* must be less than sixteen digits. During input the data item may be anywhere in the field of width *w* and its general form is:

$$\text{fixed-point number } E \text{ exponent}$$

The fixed-point number is the mantissa, and it and the exponent may each be signed or not. As might be expected, if they are not signed they are assumed to be positive. The interpretation of the numeric value of this floating-point number is that its value is that of the fixed-point number multiplied by 10 raised to the power of the exponent. This exponent is a decimal integer of either one or two digits, and if it is omitted it is assumed to be zero.

In some cases the E is omitted and a signed exponent is provided. In these cases the E is assumed to be present by implication. If the exponent is present there must be an E or a sign or both. If both the E and the exponent are omitted, the exponent is taken to be zero.

If the fixed-point number contains a decimal point on input, this will override any decimal point specification given. When no decimal point specification is given, however, the decimal point is assumed to be *d* digits in from the right-most digit of the mantissa.

During input *s* has no meaning and it may be omitted. If it is present it will be ignored. During output the internal representation of the number is put out as external data item in the form of a decimal floating-point number that is right-adjusted in a field of width *w*. Blanks are inserted on the left to fill out the field. The general form is:

$s - d$ digits. d digits E exponent

The mantissa is the fixed-point number expressed by the digits lying to the left of E. It will be unsigned if positive but marked with a minus sign if negative. The exponent is a decimal integer of one or two digits that is adjusted so that it does not have a leading 0. It will have a plus or minus sign. The numerical value of s designates the total number of digits in the mantissa. In these cases it is specifically digits and not spaces, and there is no count made of sign or decimal point. Since there are d digits to the right of the decimal point, there must be $s - d$ digits to the left. When s is omitted, it is taken to be equal to d , that is, the decimal point appears before the first digits in the mantissa.

You can note that w must be at least 5 greater than s in order to provide for the decimal point, E, the exponent sign, and two exponent digits. In cases where the mantissa is negative, w must be at least 6 greater than s . This floating-point number is interpreted to have the value of the product of the mantissa multiplied by 10 raised to the power of the exponent. Examples of floating-point constants and the format codes which describe them follow.

Constant	Format	Value
1234.1E5	E(8,1,5)	123410000.
-2.68E-1	E(8,2,3)	-0.268
.98765E5	E(8,5,5)	98765.

Character-string format codes are handled with either an A format or a P format. The P format is described later in a separate paragraph, and the general form of the A format is:

$$A(w)$$

This relates to an external representation of w characters, and there is a one-to-one representation with internal characters. During output if truncation is necessary it is done on the right, and when the output element is too short, blanks are added on the right. On output the w is often omitted, and in these cases w is assumed by the compiler to be equal to the character-string length of the element named in the data list. If for example the format notation were A(7) and the data item to be written were 'SISTERS', the output would be

SISTERS

but if the data item were 'BROTHERS', it would appear on output as

BROTHER

If the bit-string format code were B(10) and the bit string to be outputted were '011011111010', the output would be

0110111110

and the bit-string '0111' would be

0111

The general form for the *bit-string format* code is

B(*w*)

The rules governing this format code are the same as for the character-string format. When the *w* is omitted it is assumed to be the maximum length of the element on input or the current length of the item on output. During input the external data is a bit string anywhere in the field of *w*, but on output the bit string is left-adjusted in the field of *w* or truncated on the right as necessary. If padding is necessary on output, it is done on the right to fill in the field.

An item code is provided for *complex number format*. Generally speaking, complex numbers are treated as an ordered pair of real numbers, and as a result the complex format item provides for designating a pair of numbers and their form. This code applies only to numerical data and takes the following form:

C (real format item, real format item)

Each of these real format items may be an F, E, or P format. The external data consists of two adjacent fields containing the data specified by the two format items. If the second format item is omitted, it is assumed to be identical to the first.

The concept of a *picture format* item code will now be mentioned here briefly. The general form for the picture format specification for use with numeric data is as follows:

P 'numeric picture specification'

or for use with character-string data items:

P 'character picture specification'

More Format Codes

With the edit-directed format, PL/I provides an X format as a *spacing format* on input or output. The general form is:

X(*w*)

where in input this specifies that the next *w* characters of the external data are to be ignored. During input this specifies that *w* characters are to be skipped; during output this specifies that *w* characters or blanks are to be inserted into the external data. As an example, the format statement (F(3), X(15), A(8)) would specify that the data items 123 and 'TESTING' would appear on output as 123 ← 15 blanks → TESTING.

There also is available in PL/I a number of *control format items* that are applied to the spacing of input and output. The X format described above is one

of these, and some brief introduction has been given to several of the others. There are four specific others that should be discussed more fully at this point because they are *printing format items* that control actions of the output of data during printing. These are as follows:

PAGE
SKIP(*w*)
LINE(*w*)
COLUMN(*w*)

The PAGE format item causes the next output to be printed on the first line of a new page. The SKIP(*w*) format code causes $w - 1$ lines to be skipped and the next output line to be printed on the *w*th line after the last line. When *w* is zero or negative, there is a carriage return but there is no line advance, thus making it possible for characters to be over-printed. When *w* is omitted completely, SKIP(1) is assumed and no lines are skipped, but the output is put on the next line. The format item LINE(*w*) causes the next output line to be printed on the *w*th line of the current page. If *w* or more lines have been written on the page or if *w* exceeds the page limit, there is an interrupt (to be discussed subsequently) on ENDPAGE condition. The COLUMN (*w*) format item causes the first character of the next data item to be the *w*th character of the current line with blanks inserted to fill it out if necessary. If the current line already has *w* or more characters in it, a new line will be started and the first character inserted in the *w*th place of it. The COLUMN(*w*) format item is also applicable to input.

Note that SKIP and X format specify relative positioning, but LINE and COLUMN specify absolute locations. The item in parentheses, *w*, need not necessarily be an integer, since expressions to be evaluated during execution are also allowable. Such evaluation and conversion to integer values occurs when the format item is used.

The execution of the format list which contains control format items merely performs the indicated operations in the order in which they appear, and these control format items do not pair with any entry in the data list. The PAGE, SKIP, and LINE format items can be used with PUT separately as well as in format lists; additionally, SKIP can appear as an option in the GET statement.

Some examples of the use of these various control format items follow:

```
GET SKIP LIST (data list);
GET EDIT (data list) (SKIP(2), format list);
GET EDIT (data list) (SKIP(4), COLUMN(5), . . . ,
    COLUMN(40), . . . , COLUMN(1), . . .);
PUT PAGE LIST (data list);
PUT PAGE EDIT (data list) (format list);
PUT EDIT (data list) (PAGE, COL(10), . . . , SKIP(3),
    COL(10), . . . , LINE(30), COL(10), . . .);
```

When several successive format items in a format list are identical, it is not necessary to write out each one repeatedly. This reiteration may be provided by

preceding the format item code with a number indicating the number of times it is to be repeated, for example, 3 F(5,2) is identical to F(5,2),F(5,2),F(5,2). As might be expected, this value preceding the format item which indicates reiteration may be a constant, a variable, or an expression; if it is a decimal integer constant, a blank must separate the constant and the following format item. If the value of the reiteration item is zero or negative, the associated format item is skipped and not used. Sometimes this happens inadvertently, and it causes the improper pairing of data format items in the remainder of the list.

This reiteration may also be applied not only to individual items in a format list but also to items grouped within parentheses within the format list or to the entire format list, for example, 3(F(10,5),2(2 B(3),E(10,2,4))).

PL/I makes provision for record-oriented data transmission. Such transmission deals with data sets that are composed of a series of separate records. In such cases, each record is read or written as an entity, either into or from addressable buffers or into or from a specified variable. In such cases the specified variable is usually a structure or an array. The statements used are READ, WRITE, REWRITE, and LOCATE.

The use of such record-oriented data for transmission is particularly appropriate in commercial and business programming applications, and the reader who is interested in such application will find it necessary to master this type of data transmission. It is, however, beyond the scope of this brief appendix and the reader must refer to PL/I programming reference manuals for further treatment of this very important subject.

D-5. SIMPLE PROGRAMMING

PL/I has no limitations on the length of individual statements. In having this characteristic PL/I differs from many other programming languages, and thus it avoids the actual or symbolic association of a statement with a single input card. PL/I is written in free form and the computer sees a continuous stream of input. In a fashion which is similar to prose writing, a semicolon indicates the end of each PL/I statement—just as a period indicates the end of a sentence in prose. The next PL/I statement can begin in the next available location. This disassociation of statement length from a card format concept is appropriate since in many applications statements and data are introduced directly into the computer via a keyboard, a typewriter input terminal, magnetic tape, etc. Much of the layout given for PL/I statements is done simply to improve human readability.

For simplicity in presentation, all the PL/I programs given in this section will be of the format

- (1) __: PROCEDURE
 OPTIONS (MAIN); (start the main program)
- (2) GET DATA; (get data into computer)

552

(3) ____ = _____;

(4) PUT DATA (____,

____,____,____);

(5) END;

(compute formulas)

(print out answers)

(finish program)

Note that the basic program is divided into five sections.

Execution of a program begins with the operating system of the computer which *invokes* or calls the main procedure of the PL/I program. The computer then goes through the execution of the PL/I program, and after it encounters the END statement of the main procedure, the operational control of the computer is automatically transferred back to the computer's operating system. Therefore, every PL/I program can have only one main procedure. This refers to statement (1) in the simple PL/I program outline. The main procedure statement consists of a statement label, a colon, the three words **PROCEDURE OPTIONS (MAIN)**, and it terminates with a semicolon.

There must be one and only one *main* procedure in every PL/I program, and it must contain a label which is the name of the computer program. You can create your own labels for programs. As can be seen from the example statement given, the only leeway the programmer has in the program initiation statement is the selection of the program name.

The second statement in the basic program outline is a statement to get data into the computer. In the case shown it is the statement **GET DATA;**, and this is the simplest input-output mode available for use in PL/I. It is a data-directed mode and it handles self-identified information where the data item's name and its value appear together in a standard form on a standard file.

The third part of the simple PL/I program shown is concerned with making computations on the data available in the computer. The nature of these computations depends upon the specific problem to be solved. An arithmetic assignment statement is shown for illustration in the basic outline.

The fourth part of the basic program is devoted to getting answers out of the computer. The statement involved is **PUT DATA (____,____,____,____);**, and this is also a data-directed statement where the information is a list of variables in parentheses.[†]

The fifth and final portion of the basic program is the **END;** statement which indicates that the PL/I program has been completed and that control is to be returned to the operating system of the computer.

For a first PL/I program, assume that the objective is to read four test scores for a student and to calculate the average of these four test scores. Taken by itself this is a problem that is too simple for a digital computer's power if we had to calculate only the average score for a single student. The situation is different, however, if it is necessary to calculate the average score for several thousand students. The code sheet for this simple computer program, along with some trial data, is shown in Figure D-1.

[†]Some smaller computers use a subset of the general facilities available in PL/I, and the simple data-directed mode may not always be available. In these situations it is necessary to go to list-directed data and use the **GET LIST** and **PUT LIST** statements.

Figure D-1 Calculation of an average test score

A computer program consists of all the statements which are necessary to instruct a computer to solve a problem: get the data, process it, and return the results of this processing to the programmer. As a second example program, calculate the gross pay of an individual based upon his hourly pay rate and the number of hours which he is to work. The simple program for undertaking this calculation is shown in Figure D-2. The program has as its title PAY, and the program closely follows the basic outline. The arithmetic assignment statement calculates an entity entitled GROSS_PAY based upon the number of hours, NO_HOURS, which the man has worked and his PAY_RATE. GROSS_PAY is the only item received as an answer from the computer. Also shown in Figure D-2 is a typical data card; for the data card shown, the numerical value 92.34 is obtained as the answer.

It is now desirable to expand the complexity of the PL/I programs, but this should be done gradually. Assume that a certain amount of principal is given on which an interest rate is known. It is desired to calculate the amount of interest which will accrue from money held at this yearly interest rate for a prescribed number of days. For example, assume \$1,234.56 in principal which is loaned on a yearly interest rate of 8 percent; how much interest is due after 124 days? A simple PL/I program for making this type of calculation is shown in Figure D-3. Note that it still follows the basic outline with two slight modifications: comment statements have been added in the body of the program, and the formula calculations portion of the program has been expanded. Line number 5 of the coding form is the calculation which determines the amount of simple interest which is due. A second portion of the formula calculation allows the total amount of money due to be calculated by adding the amount of simple interest to the principal. The amount of simple interest and the total amount due are printed out as answers. A simple data card is shown, and the numerical answers obtained are SI = 58.71 and TOTAL DUE = 1293.27. Assume now that it is desirable to calculate the simple interest in the manner prescribed, to calculate compound interest in a similar manner, and to find the difference between the two types of interest payments. The example program of Figure D-3 illustrates the way in which to calculate simple interest. Compound interest is calculated as follows:

$$CI = \text{principal} \left[\left(1 + \frac{\text{Rate}}{N} \right)^L - 1 \right]$$

where N = number of interest periods per year

L = number of periods the money is held

The computer program for making these calculations is shown in Figure D-4.

In this section some elementary PL/I programs have been presented in a simple program format. The student should realize that the outline given is not a constraint on the manner in which he must structure his own individual programs. This should be done to suit his needs, and there are no major limitations

1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
1234567890123456789012345678901234567890123456789012345678901234567890							
1	/* EXAMPLE PROBLEM D-2 */						
2	PAY: PROCEDURE OPTIONS(MAIN);						
3	GET DATA;						
4	GROSS_PAY = NO_HOURS * PAY_RATE;						
5	PUT DATA(GROSS_PAY);						
6	END;						
7	NO_HOURS = 38 PAY_RATE = 2.43;						
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							

Figure D-2 Calculation of gross pay

1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
123456789012345678901234567890123456789012345678901234567890							
1	/* EXAMPLE PROBLEM D-4 */						
2	BOTH: PROCEDURE OPTIONS(MAIN);						
3	GET DATA;						
4	/* CALCULATE SIMPLE INTEREST */						
5	SI = PRINCIPLE * RATE * DAYS / 365;						
6	/* CALCULATE COMPOUND INTEREST */						
7	CI = PRINCIPLE * ((1 + RATE/N)**L - 1);						
8	/* FIND THE DIFFERENCE BETWEEN THE TWO FORMS OF INTEREST PAYMENTS */						
9	DIFFERENCE = CI - SI;						
10	/* PRINT OUT THE PRINCIPLE, RATE, SI, CI, AND DIFFERENCE */						
11	PUT DATA(PRINCIPLE, RATE, CI, SI, DIFFERENCE);						
12	END;						
13	PRINCIPLE = 1234.56 RATE = .08 DAYS = 360 L = 12 N=12;						
14							
15							
16							
17							
18							
19							
20							

Figure D-4 More complex interest calculations

558 on the outline of PL/I programs. There are certain conventions and idiosyncrasies which must be observed in programming, but these will be explained as they are encountered.

The remainder of this appendix is aimed toward the development of more sophisticated programming skills.

D-6. CONTROL STATEMENTS

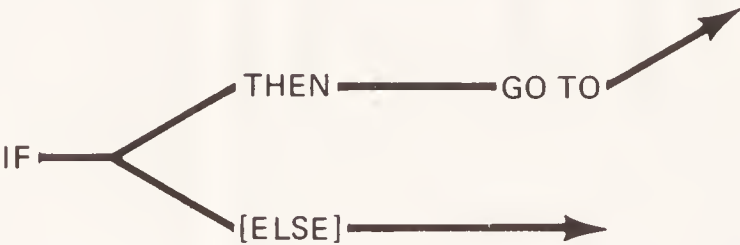
In PL/I there are three different IF statements. An IF statement is inserted in a program where an exclusive decision between alternatives must be made. The basic IF statement looks as follows:



An example of this would be as follows:

```
IF  A < B
    THEN  ANS = A + 6;
    ELSE  ANS = A;
```

The second type of IF statement has a different type of logical diagram:



In this type of IF statement one alternative causes a transfer of control to some other point in the program, and sequential execution does not continue. The ELSE need not appear. An example of this type of IF statement is as follows:

```
IF  HOURS = 0
    THEN GO TO NO_HOPE;
```

The third type of IF statement has the following logical diagram:



An example of this type of IF statement would be as follows:

```
IF  COUNT = 26
    THEN SET = 0;
```

In this type of IF statement the alternatives are “execute the THEN clause” or “do not execute the THEN clause.” In either case the next sequential statement is executed.

You should note that the word ELSE may sometimes be omitted in IF statements, but the word THEN must appear in each such statement.

IF statements can be nested one in another. In such a nesting of IF statements each ELSE clause is paired with an IF starting with the innermost level. The compiler will make the IF condition tests in the order that they are written, and as soon as it reaches a test that is not true, the checking stops and the matching ELSE clause is executed. At this time, control is then transferred out of the entire series of IF statements. An example of this would be as follows:

```
IF   PREPARE = HOPE
    THEN IF ANS = 100
        THEN SCORE = 100;
        ELSE A = B;
    ELSE A = C;
```

The DO statement in PL/I may be used to execute a group of statements or to form a loop. When the DO statement is used to execute a group of statements, it might appear as follows:

```
IF   A = B
    THEN DO;
        COST = PAY + LABOR;
        HOPE = COST * 1.50;
        END;
    ELSE . . . ;
```

A DO group is a sequence of statements whose beginning is defined by a DO statement and whose end is defined by an END statement. The more common use of the DO statement is to define and specify control for a group of statements to be used as a loop, that is, a series of statements to be executed and re-executed one or more times before control moves on to the next statement after the group.

Every DO statement must have an associated END statement to define the end of the group.

Consider the following example:

```
DO COUNTER = 1 TO 10;
statement-1
statement-2
statement-3
END;
statement-4
```

The DO and END statements specify that statements 1, 2, and 3, whatever they may be, are a DO group. The DO statement further specifies that these state-

ments are to be executed, as a group, ten times before control is transferred to statement 4. The variable COUNTER is used to control the number of times the group is executed. When the DO statement is executed for the first time, COUNTER is assigned the value 1. Statements 1, 2, and 3 are then executed. When the END statement is reached, COUNTER is incremented by one, and control is transferred back to the beginning of the group where COUNTER is tested to see that it is no larger than 10. This looping continues until the value of COUNTER exceeds 10, when control passes on to statement 4. The above example is exactly equivalent to the following:

```

                COUNTER = 1;
LOOP:   IF COUNTER > 10
                THEN GO TO NEXT;
                statement-1
                statement-2
                statement-3
                COUNTER = COUNTER + 1;
                GO TO LOOP;
NEXT:   statement-4

```

Note: Since the test is made *after* COUNTER is incremented, its value at the end of the loop will be one incrementation larger than the number of times the loop is executed. In this case, the value of COUNTER will be 11 when execution of the loop is complete.

The variable COUNTER, either as used in the DO statement or as used above, would have to be declared to represent values as great as 10 (DECIMAL FIXED(2) or BINARY FIXED(4)).

A PL/I programmer may, however, use certain variables as counters without having to declare them explicitly in a DECLARE statement. Information that is necessary for storage assignment can be assumed. This facility in PL/I is called *implicit declaration*. Any identifier whose first letter is I, J, K, L, M, or N is assumed to be a variable having BINARY and FIXED attributes, unless the same identifier is declared elsewhere to have different attributes. These assumptions, or defaults, are the same for all PL/I compilers; the precision default may vary from compiler to compiler.

The DO group in the previous example might have been written:

```

DO I = 1 TO 10;
statement-1
statement-2
statement-3
END;
statement-4

```

Unless the single letter I has been declared otherwise, it is, through its appearance in the statement, implicitly declared, by default, to be a variable with BINARY and FIXED attributes and default precision.

A variable used in this way need not be a single letter; NUMBER, for example, could be used to take advantage of implicit declaration, as could I2, I3, I4, etc. Most programmers, however, find that a single letter is more convenient.

The same letter may be used as a counter in a number of different DO loops, so long as the loops are completely separate. For example, statement 4 might be a DO statement specifying the beginning of another DO loop. The letter I could be used in that second DO statement. Although its value is 11 when the first DO loop is completed, the value is reset when the second DO statement is executed.

In the preceding DO statement, the value of I is increased by one each time the DO statement is executed. Since the variable usually serves as a counter, an increment of one is assumed. However, any increment can be stipulated. For example,

```
DO I = 2 TO 10 BY 2;
```

This DO statement causes the initial value of I to be set to two. Each time the DO statement is executed thereafter, the value is increased by two. Thus, the statements of the DO group would be executed five times, and the final value of I would be 12.

This kind of control expression might be specified when the counter is also used as a variable in an expression within the DO group. For example, the following DO group could be used to compute the volume of each of a series of circular wading pools. Assume that every pool is 12 inches deep and that the diameters range from 18 inches to 10 feet, increasing by 6 inches from size to size:

```
DO I = 9 TO 60 BY 3;  
VOL = (PI * I ** 2) * 12;  
PUT FILE (OUTPUT) LIST (I * 2, VOL);  
END;
```

The initial value assigned to I is 9, which is equal to the radius of the smallest wading pool. Each increment of 3 makes I equal to the radius of the next larger size. The volume is computed for each size, and the result is written, along with the diameter of the pool. Since I has a value equal to the radius, it must be doubled before it represents the value of the diameter. PL/I permits expressions such as $I * 2$ to be specified in a PUT statement. The expression is evaluated exactly as if it appeared in an assignment statement, and the computed value is written as indicated in the data list.

Varying values can be specified as follows:

```
DO I = 1, 8, 9, 15, 17, 25;
```

I is assigned each value in turn. After the DO group has been executed with I equal to 25, control passes to the next statement following the group. The values need not be in ascending order; they might have been written 8, 1, 15, 17, 9, 25.

The values of variables may be assigned successively to the counter in a DO statement:

```
DO X = A, B, C, D, E, F, G, H;
```



```
DIST = X * .588E13;  
PUT FILE (OUTPUT) LIST (X, DIST);  
END;
```

Assume that the variables A through H represent the light-year distance to certain stars expressed as decimal floating-point numbers. The first time the DO statement is executed, the value of A is assigned to X, and the first statement of the group, the assignment statement, is executed. The value of X is multiplied by .588E13, which is an approximation of the distance, in miles, that light travels in one year. The computed value of the expression, which is a floating-point number representing the distance to the star in miles, is assigned to DIST. Then the distance of A in light-years and the distance in miles are written as floating-point numbers. Control returns to the DO statement, and the value of B is assigned to X for the second execution of the group. Repetition continues until the distance in miles has been computed and written for the star represented by H.

Note that the letter X is used for the counter in the DO statement. As noted before, an identifier whose initial letter is I, J, K, L, M, or N can be implicitly declared to represent binary fixed-point data items. Likewise, any identifier beginning with a letter *other than* I through N can be implicitly declared to be a variable representing decimal floating-point numbers. Thus, both X and DIST can be implicitly declared merely through their appearance in this DO group (any of the variables A through H might also have been implicitly declared, although they obviously have appeared elsewhere in the program when their value was assigned).

Another method of loop control causes looping to continue as long as a certain condition exists, as follows:

```
DO WHILE (A < B);
```

Here, the values of A and B are compared each time control reaches the DO statement. The computer continues executing the statements in the DO group until the value of A becomes equal to or greater than the value of B. Obviously some computation within the DO group must alter the value of A or B, or the loop will be unending. The WHILE clause—any expression following WHILE—must be enclosed in parentheses.

Now, combining the features of both these examples, there is this form of the DO statement:

```
DO I = 1 TO 10 WHILE (A < B);
```

This causes repeated execution of the group either until the tenth execution is completed or until A no longer is less than B. As soon as either condition is satisfied, execution ceases, no matter what the status of the other.

If the programmer wants to continue execution until each of two or more conditions is met, he can write

```
DO I = 1 TO 10, 11 BY 0 WHILE (A < B);
```


The group is executed ten times. Then the new condition takes control, and execution continues until A no longer is less than B. Use of the comma to separate the control expressions has the effect of setting up successive DO statements for the same DO group (note that if the TO clause is omitted but the BY clause is included, execution could continue indefinitely).

The following statement illustrates another variation:

```
DO I = 1 TO 4, 6 TO 10;
```

The statements of the group are executed nine times. When the group is executed, the value of I will be, successively, 1, 2, 3, 4, 6, 7, 8, 9, 10. The group is not executed when I has the value 5. It is important to note that only one variable can be used as a *counter* in a single DO statement. More than one variable may appear (DO I = K TO M might be written), but only one variable may appear to the left of an equal sign. Thus, the following statement would be in error:

```
DO I = 1 TO 4, J = 6 TO 10;
```

The above would have to be written as separate DO statements, each having its own END statement.

DO groups, like IF statements, may be nested. Consider this example:

```
DO I = 1 TO 10;
statement-1
statement-2
statement-3
      DO J = 1 TO 10;
      statement-1a
      statement-2a
      statement-3a
      END;
statement-4
statement-5
statement-6
END;
```

The statements of the outer DO group—the outer DO and END and statements 1 through 6—would be executed ten times. The statements of the inner DO group—the inner DO and END and statements 1a through 3a— would be executed 100 times, ten times for each execution of the outer DO group. When the first DO statement is executed the first time, I is assigned the value 1. Then statements 1 through 3 are executed. When control reaches the second DO statement, J is assigned the value 1, and the inner loop is executed until the value of J exceeds 10. Control then passes on to statements 4, 5, and 6. When the final END statement is reached, control returns to the first DO statement. The counter I is incremented to 2, and execution proceeds through statements 1 through 3. When the second DO statement is reached for the second time, J is

reset to 1, and the inner DO group again is executed ten times before control passes to statement 4 for its second execution. The process is repeated until the outer DO group has been executed ten times. The inner DO group goes through its entire looping process immediately following each execution of statement 3.

When DO groups are nested, a *different* variable must be used for each counter. Had the same counter been used for both groups in the above example, it would reach the value of eleven when the first looping of the nested group was completed. Statements 4, 5, and 6 would be executed for the *first* time, and control would go to the next statement after the END statement of the first DO group. Although the outer loop would be executed only once, the counter would indicate that it had been executed ten times.

The example shows nesting only to the second level. Much deeper nesting is allowed; the maximum varies from compiler to compiler. In general, the maximum will be the same as that for nested IF statements.

Whatever the number of nested groups, each *contained* group will be executed to *completion* for every *single* execution of its *containing* group, just as the second group, in the above example, is executed to completion for every single execution of the outer group.

A DO statement can have a label prefix, and control can be passed to it through the use of a GO TO statement. Control can be passed to a statement within a DO group only if the group is headed by a simple DO statement, that is, a DO without any repetitive specification. Control can, however, be transferred out of a DO loop; for example, a GO TO statement might be a THEN or ELSE clause of an IF statement within the loop.

This discussion has covered only a few of the many ways DO groups can be used. Virtually the only limitation is the programmer's ingenuity.

D-7. PROCEDURES

A program may consist of a single procedure or of several procedures. During execution of the program, control can go from one procedure to another and can return to a previously executed or partly executed procedure.

A procedure is headed by a PROCEDURE statement and ended by an END statement, as follows (the dots represent the statements in the procedure):

```
UPDATE:  PROCEDURE;
          .
          .
          .
          END;
```

Each procedure must have a name, that is, each PROCEDURE statement must be labeled. A procedure name denotes an entry point through which control can be transferred to the procedure.

The division of a program into several procedures is a feature of PL/I that provides a special convenience to programmers. The procedures can be written separately, even compiled separately, and executed as a single program. A long program can be divided into logical blocks; special procedures can be written for special purposes. This feature also provides great economy in the use of internal storage space.

Control does not pass automatically from one procedure to the next. Each procedure, except the first, must be invoked, or called, separately from some other procedure. This usually occurs with the execution of a `CALL` statement, for example:

```
CALL UPDATE;
```

Execution of this statement in another procedure would transfer control to the entry point of the procedure named `UPDATE`.

The first procedure of a program must have the `OPTIONS (MAIN)` attribute specified for it in its `PROCEDURE` statement. At execution time, this procedure is called automatically to begin execution of the program.

The different procedures in a program may be entirely separate from one another, or some may be nested within other procedures. Consider the following two examples:

Example 1:

```
FIRST:  PROCEDURE OPTIONS (MAIN);
        statement-1
        statement-2
        statement-3
        statement-4
        statement-5
        statement-6
UPDATE:  PROCEDURE;
        statement-a
        statement-b
        statement-c
        END;
```

The two procedures shown are separate from one another; they are *external* procedures. All of the text of a procedure, except its entry name, is said to be contained in that procedure.

Example 2:

```
FIRST:  PROCEDURE;
        statement-1
        statement-2
        statement-3
        UPDATE:  PROCEDURE;
                  statement-a
```



```

statement-b
statement-c
END;

statement-4
CALL UPDATE;
statement-6
END;

```

In this example, UPDATE is nested within—or contained in—FIRST. FIRST is an external procedure; UPDATE is an *internal* procedure.

Execution starts with the FIRST: PROCEDURE statement. Statements 1, 2, and 3 are executed. When control reaches the UPDATE: PROCEDURE statement, that statement is ignored, and execution continues with statement 4. Upon execution of the fifth statement, CALL UPDATE, control is transferred to UPDATE. Statements a, b, and c are executed. When the END statement in UPDATE is executed, control is transferred back to the procedure, FIRST, to execute the statement immediately following the CALL UPDATE statement; that is, control is transferred to statement 6.

When UPDATE is called, it is the *invoked procedure* and FIRST is the *invoking procedure*; the CALL UPDATE statement is the *point of invocation*.

Any procedure, external or internal, can invoke another external procedure, but it cannot directly invoke an internal procedure that is contained in some other procedure. Assume that FIRST is a control procedure as follows:

```

FIRST:  PROCEDURE OPTIONS (MAIN);
        CALL A;
        CALL B;
        CALL C;
UPDATE: PROCEDURE;
        statement-a
        statement-b
        statement-c
        END;

CALL D;
CALL UPDATE;
CALL E;
END;

```

The CALL A statement invokes the procedure named A, and control is transferred to A. When it returns to FIRST, after completion of A, the statement CALL B is immediately executed, and control is transferred to the procedure B, later to return to the CALL C statement in FIRST. Execution of FIRST continues in this way.

Any of the procedures invoked by FIRST might invoke other procedures. For example, a statement in procedure A might invoke a procedure Z. But control eventually returns to the statement in the invoking procedure that immediately follows the point of invocation (from Z to A and from A to FIRST).

Since UPDATE is an internal procedure contained in FIRST, FIRST is the only external procedure that can invoke it. If another procedure, say procedure F, were contained in UPDATE, only UPDATE could invoke it. No statement in FIRST could call F.

The following example demonstrates how an internal procedure can be made available to other external procedures:

```

PROC_1:  PROCEDURE;
        statement
        .
        .
        .
BETA:    ENTRY;
        CALL PROC_A;
        PROC_A: PROCEDURE;
            statement
            .
            .
            .
        END;
    END;

```

In this example, PROC_1 has two entry names, PROC_1 and BETA. The statement labeled BETA is an *ENTRY statement*. An ENTRY statement specifies that its label can be used in a CALL statement to invoke the procedure at the point where the ENTRY statement appears. An ENTRY statement declares its label to have the ENTRY attribute in the same way that a PROCEDURE statement declares its label to have the ENTRY attribute. PROC_1 can be invoked at its beginning with a CALL PROC_1 statement, or at the ENTRY statement with a CALL BETA statement (an ENTRY statement can be given any valid label to be used in an invoking call).

When PROC_1 is invoked at BETA, the first statement executed invokes PROC_A; the invoking procedure has, in effect, called an internal procedure outside its normal scope. When execution of PROC_A is completed, control returns to the statement in PROC_1 that follows the point of invocation, in the preceding example, the END statement of PROC_1. Consequently, control returns to the procedure that invoked PROC_1 at the entry point, BETA.

When PROC_1 is invoked at its primary entry point, at PROC_1, the statements are executed in the order indicated by the programmer. When control encounters the ENTRY statement, it skips to the next statement, in this case, the CALL PROC_A statement.

More than one procedure may be contained in a single procedure, either as separate internal procedures or as nested internal procedures:

```

PROC_1:  PROCEDURE;
        statement-1
        statement-2

```

```
CALL PROC_A;
PROC_A: PROCEDURE;
    statement-1a
    statement-2a
    CALL PROC_B;
    PROC_B: PROCEDURE;
        statement-1b
        statement-2b
        statement-3b
    END;
    statement-4a
    statement-5a
    statement-6a
END;
statement-4
statement-5
CALL PROC_C;
PROC_C: PROCEDURE;
    statement-1c
    statement-2c
    statement-3c
END;
statement-7
statement-8
statement-9
END_1: END;
```

PROC_A, PROC_B, and PROC_C are all contained in PROC_1. PROC_B also is contained in PROC_A. PROC_1 can invoke either PROC_A or PROC_C; either PROC_A or PROC_C might invoke one another; but only PROC_A can invoke PROC_B.

A contained procedure must be wholly within its containing procedure; all of the statements of a contained procedure must appear between the PROCEDURE and END statements of the procedure in which it is contained.

A GO TO statement may be used in an internal procedure to transfer control from that procedure to any labeled statement (except entry points) in any containing procedure. A GO TO statement, generally, cannot be used to transfer control between separate procedures or from an outer procedure to a statement in any of its contained procedures.

In the previous example, a GO TO statement in PROC_B could transfer control to a statement in PROC_A or PROC_1 but not in PROC_C because PROC_B is not contained in PROC_C. A GO TO statement could also be used to transfer control from PROC_A or from PROC_C to a statement in PROC_1. A GO TO statement cannot be used to transfer control to a statement in

PROC_B, or PROC_C; PROC_1 can use a GO TO statement only to transfer control from one to another of its nine internal statements or to the labeled END statement.

Control returns to an invoking procedure when the END statement of an invoked procedure is reached. Often, there are reasons why a programmer wants control to return *before* the END statement is reached. The procedure UPDATE illustrates such a situation.

The execution of UPDATE will end after the end-of-file condition arises for MASTER file. With UPDATE used as one of several procedures in a program, another ON statement would be required. The END statement of the UPDATE procedure could be given a label, say ENDUP, and the following statement could be inserted somewhere in the program:

```
ON ENDFILE (MASTER) GO TO ENDUP;
```

After the last record in MASTER file is read and processed, control is transferred to the END statement and is returned to the invoking procedure.

Another method of returning control from an invoked procedure to its invoking procedure is handled through the use of the RETURN statement. When a RETURN statement is executed, the result is the same as if the procedure had been completely executed; control returns immediately to the first statement following the point of invocation. A RETURN statement could not be used in the preceding ON ENDFILE statement because a RETURN statement cannot be the contained statement of an ON compound statement.

A RETURN statement might appear in a procedure as follows:

```
IF CNTR = 100  
  THEN RETURN;  
GO TO LOOP;
```

Presumably, some operation in the procedure is to be done 100 times and a counter is used, rather than a DO loop, to control the number of repetitions.

In general, each procedure should be written in such a way that it is as independent as possible. It usually is preferable to return control to an invoking procedure with the use of the END statement or a RETURN statement, even if a GO TO statement might be used. It also is preferable, so far as is possible, to keep all references to a single variable within the same procedure (or procedures contained in it). The reasons for this will be made apparent in the following discussions of "Program Execution" and "Recognition of Names."

Program Execution

Any name that represents a data item actually represents the location in storage where that data item is recorded. The compiler analyzes the attributes of a variable to determine the amount of storage area that is needed and *when* it will have to be available.

The fact that certain variables are used in one part of a program and not used in others makes it possible to set aside—or *allocate*—the same physical storage location, at different times, to different variables.

If during execution of a single procedure, but nowhere else in the program, a 100-character field is required for the variable TABLE, the space need not actually be allocated until execution of that procedure begins. If TABLE is not referred to again, there is no need to keep the space reserved after execution of the procedure is completed. The storage area can be used for other purposes.

Such dynamic control of storage is called *automatic* allocation. Variables declared in a procedure are assumed to have the AUTOMATIC storage class attribute; they are allocated when the procedure is invoked and are freed when it is completed.

Some variables may be declared to have the STATIC storage class attribute; they are allocated when the *program* execution begins, and they remain allocated throughout execution of all the procedures in the program.

A third storage class is the *controlled* storage class. Variables declared to have the CONTROLLED attribute are allocated upon execution of an ALLOCATE statement referring to the variable name. The storage allocated for a controlled variable is released upon execution of a FREE statement referring to the variable. Consider the following example:

```

      DECLARE A (50, 50) CHARACTER (15)
              CONTROLLED;
      .
      .
      .
      ALLOCATE A;
      .
      .
      .
      FREE A;
      .
      .
      .

```

In the DECLARE statement, A is declared to be a two-dimensional array of character strings having the CONTROLLED attribute. Storage is not assigned for A until execution of the ALLOCATE A statement. Upon execution of the FREE A statement, storage no longer is allocated for A; the storage can be used for other purposes.

As long as a procedure is active, its automatic variables remain allocated. They are not released until the procedure is terminated. The first, or MAIN, procedure remains active throughout the entire program.

A procedure is activated whenever it is invoked through any of its entry points—the procedure name or another entry name. A procedure can be termi-

nated in any of several ways: when its END statement is executed, when a RETURN statement in the procedure is executed, when a STOP statement is executed in *any* procedure, or when a GO TO statement transfers control to an outer procedure.

Figure D-5 shows a portion of a program that consists of at least three external procedures, FIRST, THIRD, and SIXTH (SIXTH is not illustrated). One procedure, SECOND, is contained in FIRST; and two procedures, FOURTH and FIFTH, are contained in THIRD, with FIFTH also contained in FOURTH. The numbers at the left are merely for reference; they are not part of the program. FIRST is activated automatically. When statement 1 is executed, all automatic variables declared in FIRST are allocated. Although SECOND is contained in

```

1      FIRST:  PROCEDURE OPTIONS (MAIN);
2              CALL SECOND;
3              SECOND:  PROCEDURE;
4                  statement
5                  statement
6                  statement
7                  END;
8              CALL THIRD;
9              CALL SIXTH;
10             END;
11      THIRD:  PROCEDURE;
12              statement
13              CALL FOURTH;
14              statement
15              statement
16              FOURTH:  PROCEDURE;
17                  statement
18                  statement
19                  IF M = N THEN RETURN;
20                  CALL FIFTH;
21                  FIFTH:  PROCEDURE;
22                      statement
23                      IF M = N THEN
24                          GO TO A;
25                      statement
26                      END;
27                  IF M = N THEN STOP;
28                  END;
29      A:      statement
                END;

```

Figure D-5 Termination of blocks

FIRST, it is not activated until it actually is invoked; only its *entry name* is made available when FIRST is activated.

Statement 2 invokes SECOND, and allocations for it are made. Then its statements are executed. When the END statement in SECOND (statement 7) is executed, SECOND is terminated, its automatic allocations are released, and control returns to FIRST. *SECOND is terminated through execution of its END statement.* Storage locations that were automatically allocated for SECOND are now available for use by other procedures.

Statement 8 invokes THIRD. FIRST remains active (as it will throughout execution of the entire program), and THIRD is activated. Again, the internal procedures are not activated, except that the entry name of FOURTH is made available to THIRD.

Figure D-5 shows why a GO TO statement cannot be used to transfer control to a labeled statement in an inner procedure. Since the inner procedures are not active, statement labels within them are not available. Even the entry name FIFTH is not available to THIRD. When an inner procedure *is* active, however, its *outer* procedures also are active; consequently, control can go to a labeled statement in an outer procedure.

Statement 13 causes the activation of FOURTH (THIRD remains active, because it has not been terminated). If the THEN clause of statement 19 is executed, control returns to statement 14; FOURTH is terminated before it activates FIFTH. In such a case, *FOURTH is terminated through execution of a RETURN statement.*

If the comparison in statement 19 reveals that M is not equal to N, FIFTH is activated. At this point, all but one of the procedures illustrated are active. Only SECOND has been terminated (SIXTH, of course, and any procedures it might call have not yet been activated).

If the THEN clause in statement 23 is executed, both FIFTH and FOURTH are terminated, and control is transferred to statement 28, the statement in THIRD that is labeled A. *FOURTH and FIFTH are terminated when the GO TO statement transfers control to an outer procedure.*

However, if M still is not equal to N, statements 24 and 25 are executed. FIFTH is terminated normally, and control returns to FOURTH.

If the STOP statement—the THEN clause of statement 26—is executed, program execution ceases. *All of the procedures are terminated when the STOP statement is executed.*

Otherwise, FOURTH is terminated normally, execution returns to statement 14, and the remaining statements of THIRD are executed. When THIRD is terminated by execution of its END statement, FIRST is the only procedure that remains active. It immediately activates SIXTH. FIRST remains active until control returns to execute its END statement. Execution of the END statement of the MAIN procedure has the same effect as execution of a STOP statement; program execution ceases.

Before control returns from SIXTH, however, other procedures could be

activated; SIXTH might invoke internal procedures contained in it, or it might invoke other external procedures. *Even THIRD might be reactivated.*

It is important to note that if THIRD were to be reactivated, its variables would not represent the same values that they represented before the previous termination. When allocation for a variable is released, its value no longer can be determined.

D-8. RECOGNITION OF NAMES

Questions arise as to when names can be referred to in a procedure. PL/I allows some names to be referred to in any procedure of the program, regardless of the procedure in which the appropriate DECLARE statement might be located. Such names are *known* throughout the program, that is, their *scope* is the entire program. Used in this way, known implies that a reference to a name is valid, and scope indicates the range of the program throughout which the name is known. The scope of a name is affected by the way in which the name is declared, the attributes declared for it, and the procedure in which it is declared.

Declaration of a name takes place at the time of compilation. It is clear that the compiler must have complete information about every name that appears in a program before it is possible for the compiler to translate the program statement into machine language. All or part of this information which the compiler needs may be based on assumptions made by the compiler. In PL/I it is possible for an identifier to be *explicitly* declared to have certain attributes in a DECLARE statement. It is also possible for an identifier to be *contextually* declared. If an identifier appeared with a statement as a label prefix, it is contextually declared to be a statement label; an identifier that is referred to in a CALL statement would be assumed to be an entry name; an identifier that appears in an input-output statement in a context in which only a file can appear is assumed to have the FILE attribute; and so on.

The PL/I compiler is also written so that identifiers are *implicitly* declared. Any identifier, except language words appearing in their proper context, is assumed to be a variable representing arithmetic data by the PL/I compiler. Such data is recorded either in decimal floating-point form or in binary fixed-point form, depending upon which letter of the extended alphabet appears as the initial character of the identifier. An identifier is given BINARY and FIXED attributes if its initial character is I, J, K, L, M, or N. The identifier is given the DECIMAL and FLOAT attributes in all other cases.

It is quite important to gain a fuller understanding of the recognition of PL/I names, and also the attributes and scope of names. In addition it is necessary that the programmer understand where his names are known. More complete treatments of these important subjects are given in standard PL/I reference manuals.

So far in this appendix, each data name in discussions and examples has been shown to refer to a single item of data.

In PL/I, data can be named so that an entire collection of data items can be referred to by a single name; certain subcollections can be referred to by another name, or individual elements can be referred to separately.

There are two kinds of data collections to which collective names can be given, *structures* and *arrays*. The major difference between the two is in the kind of data an array name or a structure name represents. An array is a table of homogeneous data items; each item has the same attributes as have all the other items. In an array of arithmetic items, for example, all of the items have the same base, scale, and precision; in a character-string array, all of the items have the same length. The elementary data items of a structure, on the other hand, need not be of the same data type nor have the same attributes. Some names in a structure might refer to string data while other names in the same structure could refer to arithmetic data.

Collective naming does not alter the data in any way. Data in a structure or an array is no different from what it would be if it were referred to by a single variable. Collective naming merely gives a programmer more convenience in referring to and manipulating data.

Structures

A structure is a graded system of names that refers to an area of internal storage. At the first level, a single name—the *major structure name*—refers to all of the data items stored in the entire area that is allocated to that structure. At the second level, certain portions of the area are renamed. This renaming continues until, at the deepest level, a single name, at any particular point in time, refers to a single data item; at the deepest level, in fact, each name is a variable, as previously defined.

Consider a program to figure a weekly payroll. Once employee, John J. Doe, whose pay number is 68584, works 40 hours of regular time and 5 hours of overtime. He is paid \$4.00 per hour for regular time and \$6.00 for overtime.

His weekly pay record, with all of the above information, is read and assigned to a structure named PAYROLL. The information could be ordered:

DOE JOHN J 68584 40 05 400 600

If this were referred to merely by the name PAYROLL, it might be treated as a character string; but, as a character string, it would be difficult to get to individual items within the string, and arithmetic operations would involve conversion. However, a name can also be given to each element. The names for John Doe’s pay record, and the data each name represents might, conceptually, look like this:

	LASTNAME	DOE
	FIRSTNAME	JOHN
	MIDDLENAME	J
PAYROLL	PAY_NO	68584
	REGLRHOURS	40
	OVTIMHRS	05
	STRATE	400
	OVRTIMRATE	600

Thus, a programmer can refer to the entire collection of data items by the name, PAYROLL, or he can refer to an individual item by an individual name.

It often is valuable to be able to refer collectively to more than one, but not all, of the variables in a structure. The names of the variables suggest an intermediate classification:

		LAST	DOE
	NAME	FIRST	JOHN
		MIDDLE	J
	PAY_NO		68584
PAYROLL			
	HRS	REGLR	40
		OVTIM	05
		STRATE	400
	RATE	OVRTIM	600

The major structure, PAYROLL, contains the substructures, NAME, HRS, and RATE. PAY_NO is not a substructure because it represents only a single data item.

During execution of a program in which PAYROLL is used, a number of different pay records would be read and assigned to PAYROLL. The corresponding elements of the records would each be assigned to its proper variable.

A reference to any structure or substructure is a reference to all of the data items referred to by the elementary names in that structure or substructure. It is, perhaps, clearer to say that a reference to any name is really a reference to the *names* at the next deeper level. That is, a reference to the name at the first level (PAYROLL, in the example) is a reference to all of the names at the second level (NAME, PAY_NO, HRS, RATE). A reference to a name at the second level (NAME) is a reference to all of the names in that substructure at the third level (LAST, FIRST, and MIDDLE).

Since there are no levels deeper than LAST, FIRST, and MIDDLE, they are not substructures. They are elementary names, like PAY_NO.

When a structure is declared, the level of each name is indicated by a *level number*. The major structure name, at the first level, always is given the level

number 1. Each name at a deeper level is given a greater number to indicate the level depth. For example:

```
DECLARE    1 PAYROLL, 2 NAME, 3 LAST,
           3 FIRST, 3 MIDDLE, 2 PAY_NO,
           2 HRS, 3 REGLR,
           3 OVTIM, 2 RATE, 3 STRATE, 3 OVRTIM;
```

The same DECLARE statement could be written as follows:

```
DECLARE    1 PAYROLL,
           2 NAME,
             3 LAST,
             3 FIRST,
             3 MIDDLE,
           2 PAY_NO,
           2 HRS,
             3 REGLR,
             3 OVTIM,
           2 RATE,
             3 STRATE,
             3 OVRTIM;
```

The order of the appearance of names in a DECLARE statement, along with their level numbers, determines the structuring. Except for the major structure name, no particular numbers must be specified. The only requirement is that deeper level names must have greater numbers. The major structure name must be declared with the level number 1, and it must be the first name listed in the structure declaration. It must be followed by one of the names at the second level. Each name at the second level must have a level number greater than 1, and if it is a substructure, it must be followed by the names within that substructure. Each substructure is completely declared before the next substructure declaration begins, and each substructure must have a level number equal to or less than the level number of the immediately preceding substructure at the same level. For example, if RATE were declared with the level number 3, it would be considered to be part of the substructure HRS.

The level numbers of the above structure might have been declared in this way:

```
DECLARE    1 PAYROLL,
           8 NAME,
             20 LAST,
             20 FIRST,
             20 MIDDLE,
           6 PAY_NO,
           2 HRS,
```

```

        3 REGLR,
        3 OVTIM,
    2 RATE,
        8 STRATE,
        8 OVRTIM;

```

Exactly the same structuring would result.

The number of levels allowed within a structure will vary from compiler to compiler.

When a structure is declared, attributes may be specified for the elementary item names; otherwise, default attributes would apply. Attributes can be factored in a structure declaration. For example:

```

DECLARE  1 PAYROLL,
        2 NAME,
            3 LAST CHARACTER(12),
            3 FIRST CHARACTER(8),
            3 MIDDLE CHARACTER(1),
        2 PAY_NO CHARACTER(5),
        2 HRS,
            (3 REGLR,
             3 OVTIM)
            FIXED DECIMAL(2),
        2 RATE,
            (3 STRATE,
             3 OVRTIM)
            FIXED DECIMAL(3,2);

```

Although level numbers *must* appear with a name in the DECLARE statement, they do not appear with the name in any other reference to it.

Qualified Names. All names within a single procedure must be unique. But within structures, it is often convenient to be able to use the same identifier for related names. In the above structure, for example, it would be convenient to refer to the items in HRS and RATE as “regular hours” and “regular rate” and “overtime hours” and “overtime rate.”

In fact, the elements can be given the same names. The last portion of the structure might be declared:

```

    2 HRS,
        3 REGLR,
        3 OVRTIM,
    2 RATE,
        3 REGLR,
        3 OVRTIM;

```

The use of a *qualified name* in referring to the individual item avoids ambiguity.

A qualified name is a substructure or element name that is made unique by qualifying it with one or more names at a higher level. The individual names within a qualified name are separated by a period. The above items could be referred to by the following qualified names:

HRS.REGLR
 RATE.REGLR
 HRS.OVRTIM
 RATE.OVRTIM

Any of the names in PAYROLL, except PAYROLL itself, need not be unique within the procedure in which it is declared.

They all could be qualified. For example:

PAYROLL.NAME
 PAYROLL.NAME.LAST
 or NAME.LAST
 or PAYROLL.LAST

All of the qualifying names need not appear. A name need be qualified only so far as is necessary to make it unique.

Structure Expressions. Structure and substructure names can be used in arithmetic or string expressions.

For example, consider the following statement:

PAY = HRS * RATE;

PAY, in this case, must be a structure name referring to two arithmetic variables. The above statement would have the same effect as the following two statements:

PAY.REGLR = HRS.REGLR * RATE.REGLR;
 PAY.OVRTIM = HRS.OVRTIM * RATE.OVRTIM;

A reference to a structure or substructure name in an expression is a reference to all of the elements within that structure or substructure. In the case of an arithmetic expression, the expression is evaluated separately for each set of corresponding elements.

A structure expression need not be limited to structure names. The expression HRS/5 would be valid and it would give the same result as the two expressions, HRS.REGLR/5 and HRS.OVRTIM/5.

Summary. A structure is a hierarchy of names built upon a list of individual variables, so that the variables can be referred to singly or in groups or so that the entire list of variables can be referred to by a single name.

Structuring has no effect upon the data themselves; it merely provides greater flexibility in referring to the data.

Arrays

An array is a named table of data items all of which have identical attributes. Only the array, itself, is given a name. An individual item of an array is referred to by giving its location within the array. The location is specified by a *subscript* following the array name.

Assume TABLE has been declared to be an array of twelve elements. TABLE (1) refers to the first item in the list, TABLE (2) to the second, TABLE (3) to the third, etc. Each of the numbers, (1), (2), or (3), is a subscript that gives the location, within TABLE, of a particular data item.

An array name is declared in a DECLARE statement by giving its name, the number of elements in the array, and the attributes of the items.

```
DECLARE TABLE(12) DECIMAL FIXED(2);
```

This specifies that the name, TABLE, refers to an array of twelve data items, each of which will have a value that can be represented by two decimal digits. TABLE is declared to have the *dimension attribute* of (12). The *bounds* of the dimension are 1 and 12. In any declaration, the dimension attribute must immediately follow the array name. Assume that the following numbers have been assigned to TABLE:

- 31
- 43
- 42
- 57
- 64
- 73
- 79
- 79
- 69
- 58
- 49
- 40

Thus, TABLE(1) would refer to the data item 31, TABLE(6) to 73, TABLE(12) to 40. The expression TABLE(7) + TABLE(1) would yield a value of 110.

Assume that the values assigned to TABLE represent the average temperature of the months of a particular year. TABLE(1) is the January average, TABLE(2) is the February average, etc. As TABLE was declared in the previous DECLARE statement, the data items could be referred to singly or as a whole. For various reasons, a programmer might want to consider the year as divided into quarters; it might be convenient to be able to use a single reference to all of the average temperatures of a quarter of a year. He might declare TABLE as follows:

```
DECLARE TABLE(4,3);
```

In this statement, the dimension attribute specifies that TABLE refers to twelve data items, but that TABLE is considered to consist of four lists of three items each. It has two dimensions, one with bounds of 1 and 4, the other with bounds of 1 and 3. The data might be recorded in storage in exactly the same way as with the first declaration, but *conceptually* it is ordered differently.

Following are two different ways in which the arrangement might be conceptually illustrated. The one on the left treats it as four consecutive lists of three items each; the one on the right treats it as a matrix of four rows and three columns:

	TABLE(1,1)	TABLE	(n,1)	(n,2)	(n,3)
31 }					
43 }	(1,2)	(1,m)	31	43	42
42 }	(1,3)	(2,m)	57	64	73
57 }	(2,1)	(3,m)	79	79	69
64 }	(2,2)	(4,m)	58	49	40
73 }	(2,3)				
79 }	(3,1)				
79 }	(3,2)				
69 }	(3,3)				
58 }	(4,1)				
49 }	(4,2)				
40 }	(4,3)				

With such a two-dimensional array, a programmer can refer to cross sections of the array (that is, to all of the variations within the bounds of one dimension) by writing an asterisk in place of that dimension in a subscripted name. For example,

```
PUT LIST (TABLE(1,*));
```

The above statement would have exactly the same result as the following:

```
PUT LIST (TABLE(1,1), TABLE(1,2), TABLE(1,3));
```

In the example below, the first statement would have exactly the same effect as the *four* statements that follow it:

```
A = TABLE (*,2) * 2;  
A(1) = TABLE(1,2) * 2;  
A(2) = TABLE(2,2) * 2;  
A(3) = TABLE(3,2) * 2;  
A(4) = TABLE(4,2) * 2;
```

Note that A must be an array of four elements, since the expression TABLE (*,2) * 2 would, using TABLE as declared previously, result in four separate values.

In whatever concept one chooses to visualize an array, it is important to remember that although the dimension attribute in the preceding example

specifies a two-dimensional array, the location of each item in storage could be exactly the same as if it were a one-dimensional or three-dimensional array. It could, in fact, actually be declared as a three-dimensional array with the following DECLARE statement:

```
DECLARE TABLE(2,3,2);
```

Note that the number of specifications, separated by commas, is the same as the number of dimensions, and that the product of the numbers is equal to the number of items in the array: (12), (4,3), (2,3,2).

Using the same data, TABLE(2,3,2) might be envisioned as follows:

31	{	TABLE(1,1,1)	TABLE	(1, n,m)	(1, $n,1$)	(1, $n,2$)
43		(1,1,2)				
42		(1,2,1)		(1,1, m)	31	43
57		(1,2,2)		(1,2, m)	42	57
64		(1,3,1)		(1,3, m)	64	73
73		(1,3,2)				
79		(2,1,1)				
79		(2,1,2)				
69		(2,2,1)				
58		(2,2,2)	TABLE	(2, n,m)	(2, $n,1$)	(2, $n,2$)
49		(2,3,1)				
40		(2,3,2)		(2,1, m)	79	79
			(2,2, m)	69	58	
			(2,3, m)	49	40	

The dimension attribute (2,3,2), specifies that TABLE represents a list of twelve data items and that the list will be referred to as if it consists of two sublists, each of which is further divided into three sublists of two items each.

The same result could be achieved through declaring TABLE as a structure.

```
DECLARE 1 TABLE,
      2 TABLE_1,
      3 TABLE_1_1,
      4 TABLE_1_1_1,          (31)
      4 TABLE_1_1_2,          (43)
      3 TABLE_1_2,
      4 TABLE_1_2_1,          (42)
      4 TABLE_1_2_2,          (57)
      3 TABLE_1_3,
      4 TABLE_1_3_1,          (64)
      4 TABLE_1_3_2,          (73)
      2 TABLE_2,
      3 TABLE_2_1,
      4 TABLE_2_1_1,          (79)
      4 TABLE_2_1_2,          (79)
```

3 TABLE_2_2,	
4 TABLE_2_2_1,	(69)
4 TABLE_2_2_2,	(58)
3 TABLE_2_3,	
4 TABLE_2_3_1,	(49)
4 TABLE_2_3_2;	(40)

The actual data locations referred to could be the same, no matter how the list is referred to.

Declaring a three-dimensional array for the monthly temperature averages for one year means that the year is considered as two six-month periods, with each six-month period further divided into three two-month periods.

A more practical application of a three-dimensional array, used in connection with monthly temperature averages, would be one that contained all the monthly averages over a period of years, say ten years. The averages for each year probably would be referred to as in TABLE(4,3), and the third dimension of the array would vary through the ten years. It could be declared as follows:

```
DECLARE TABLE(10,4,3);
```

If the list contained temperatures for the years 1951 to 1960, the average temperature for July 1957 would be referred to as TABLE(7,3,1), or the seventh year, the third quarter, the first month.

A four-dimensional array might be specified for a monthly average temperature table that spanned a century. The century could be divided into decades, each decade into years, each year into quarters, and each quarter into months.

```
DECLARE TABLE(10, 10, 4, 3);
```

If the above were declared, the subscripted name, TABLE(6,2,3,1), would represent the first month of the third quarter of the second year of the sixth decade.

The limit to the number of dimensions that can be declared for an array is specified for each PL/I compiler.

If the average temperatures were available over a period of two centuries, TABLE might be declared as follows:

```
DECLARE TABLE(2, 10, 10, 4, 3);
```

There would be 2,400 items in this five-dimensional array (2 × 10 × 10 × 4 × 3).

Think of the rightmost number of any dimension attribute as specifying a one-dimensional array. The next number to the left specifies the number of one-dimensional arrays that make up a two-dimensional array. The next number to the left specifies the number of two-dimensional arrays that make up a three-dimensional array, etc. In the declaration of TABLE as a five-dimensional array, the leftmost number (2) specifies the number of four-dimensional arrays that make up the five-dimensional array.

Variable Subscripts. Subscripts in a name used to refer to an element of an array need not be numbers. Any variable having an arithmetic integer value can be used in place of any of the numbers of a subscript. For example, if ALPHA has the value 2 and BETA has the value 3, the expression A(ALPHA, BETA) is the same as the expression A(2,3).

The ability to use variables as subscripts is especially valuable in DO groups in which array data is manipulated. For example, the statements below specify that the average temperature be written for the month of February of each year of a century (assume that the average temperatures have been recorded for every month of every year since January 1865; assume further that the variable YEAR has been declared with the attributes DECIMAL FIXED(4) and has been given an initial value of 1865):

```
DO  I = 1 TO 10;  
    DO J = 1 TO 10;  
        PUT LIST (YEAR, TABLE(I,J,1,2));  
        YEAR = YEAR + 1;  
    END;  
END;
```

When the first DO statement is executed, I is set equal to 1, and control passes to the second DO statement, where J is set equal to 1. When the PUT statement is executed the first time, both I and J in the expression, TABLE(I,J,1,2), have the value 1. The year 1865 and the temperature for February 1865 are written on the standard output file. YEAR is made equal to 1866, the END statement is executed, and control returns to the innermost DO statement. J is incremented, and the operation is repeated. When the inner DO group has been executed ten times, control returns to the outer DO statement. I is incremented, and control again passes to the inner DO statement. J is reset to 1, and the loop is executed ten more times. This time, the value of I is two; the temperatures of the second decade are written. The statements are executed until TABLE(10,10,1,2) has been written.

Array Expressions. A single element of an array of arithmetic data is a single arithmetic data item. It can be manipulated in the same way as any other such item.

Likewise, entire arithmetic arrays (or cross sections of them) can be used in arithmetic expressions. Such expressions are evaluated element by element, and a value is returned for each element. Expressions containing array names are *array expressions*, and they always return an array value.

Note: Although an array may be conceived as being similar to an arithmetic matrix, array expressions are not always expressions of conventional matrix algebra.

Since array expressions are always evaluated on an element-by-element basis, all arrays referred to in a single array expression must have identical dimensions and bounds.

If A is the array:	3	5	8
	— 2	7	6
	5	— 6	4
Then —A is the array:	— 3	— 5	— 8
	2	— 7	— 6
	— 5	6	— 4
A * 2 is the array:	6	10	16
	— 4	14	12
	10	—12	8
If B is the array:	2	1	3
	4	2	1
	8	— 2	— 3
Then A + B is the array:	5	6	11
	— 2	9	7
	13	— 8	1
A * B is the array:	6	5	24
	— 8	14	6
	40	12	—12

Summary

An array is a list of homogeneous elements that are referred to by subscripted names. The subscript refers to the location of the element within the list.

The dimensions declared for an array are merely an indication of how elements will be referred to; they do not indicate any particular arrangement of the actual data items in the storage area.

All of the arrays in the discussion and in the examples have been arithmetic arrays. PL/I allows arrays of string data, either character-string or bit-string; all elements must be of equal length. A programmer may even declare a statement-label array, in which each element refers to a statement label. Arrays may be elements of a structure. And, in fact, an array may be an array of structures, in which each element of the array is an entire structure. All of the structures of an array of structures must be identical.

Discussions of these other types of arrays can be found in other PL/I publications.

D-10. ARGUMENTS AND PARAMETERS

In PL/I the names of variables, files, etc., can be referred to in procedures only when the procedure lies within the scope of the name. PL/I does make provision, however, for the scope of a name to be extended to a procedure in which the

name would normally not be known. This is done by *passing* the name as an *argument of the invocation*. For example consider the following program:

```
P1:  PROCEDURE;
      DECLARE PAY_ _ _ _;
           TABLE_ _ _ _;
           HOURS_ _ _ _;
      .
      .
      .
      CALL P2(PAY, TABLE, HOURS);
      .
      .
      .
      END P1;
```

In this example, the list of names after CALL P2 is an *argument list* which is enclosed in parentheses, and the individual names are separated by commas. In invoking the procedure P2 the list of names is passed to the procedure. Procedure P2 is a *subroutine* and, for example, P2 might appear

```
P2:  PROCEDURE(A,B,C);
      .
      .
      .
      END P2;
```

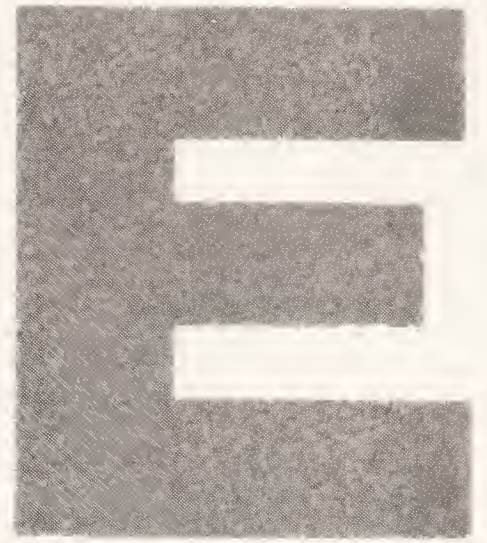
In this case A,B,C are contextually declared to be *parameters*. It is clear that a parameter is a name used in a procedure to represent another name or some other expression that is passed to that procedure as an argument of the invocation. The name is contextually declared to be a parameter through its appearance in the parameter list of a PROCEDURE or ENTRY statement.

Through the use of arguments and parameters, subroutines and functions can be used throughout a program to perform identical operations upon many different data items whose names may be known only within the invoking procedure. A subroutine or function need not lie within the scope of a name that is referred to within the subroutine or function. The passing of the name as an argument of the invocation effectively extends the scope of the name through the procedure that is invoked.

An argument must be passed by each parameter appearing in the parameter list of the invoking procedure, and the arguments must appear in the same order as the corresponding parameters in the parameter list.

Since the name (not the value itself) is passed, storage need not be allocated for each parameter that represents a data variable. It should be noted, however, that attributes must be declared for each parameter (explicitly, contextually, or implicitly) within the subroutine or function.

This appendix has presented a very brief summary of the features of the PL/I language. There are, of course, many excellent textbooks and reference manuals which present the language more completely for those who desire to study it comprehensively.



OPERATIONS ON BINARY, OCTAL, AND HEXADECIMAL NUMBERS

The objective of this appendix is first to present methods for converting from decimal to binary, octal, or hexadecimal, and vice versa and then to discuss the addition and subtraction of numbers in these systems. While many experienced programmers seldom encounter the need for such operations, on some systems the location of execution-time “bugs” unfortunately necessitates a limited facility with these operations. As the quality of run-time diagnostics continues to improve, the necessity to perform these operations will appear less frequently and hopefully will in time be eliminated for all but the system programmers. For this reason, we have chosen to place this material in an appendix.

E-1. CONVERSION FROM DECIMAL TO BINARY

To convert an integer number in the decimal system to a number in the binary system, an algorithm involving successive divisions by 2 can be used. The steps are:

1. Divide the decimal number by 2, noting the quotient and the remainder.
2. The remainder becomes a digit in the binary number.
3. If the quotient is nonzero, proceed from Step 1, using the quotient of the decimal number.
4. If the quotient is zero, we are finished.

The remainder of the first division becomes the rightmost binary digit; remainders of successive divisions occupy successive positions to the left. Conversion of the decimal 17022 to binary is accomplished as follows:

	Quotient	Remainder
17022/2	8511	0
8511/2	4255	1
4255/2	2127	1
2127/2	1063	1
1063/2	531	1
531/2	265	1
265/2	132	1
132/2	66	0
66/2	33	0
33/2	16	1
16/2	8	0
8/2	4	0
4/2	2	0
2/2	1	0
1/2	0	1

The binary number is 100 001 001 111 110.

Decimal fractions are converted to binary by successive multiplications. The steps are as follows:

- 1. Multiply the decimal fraction by 2.
- 2. The integral part becomes the digit.
- 3. If the fractional part is nonzero, repeat from step 1 until either (a) a zero fraction is obtained or (b) sufficient digits have been computed.

The integral part resulting from the first multiplication is the first digit to the right of the decimal point, which is more appropriately called the *binary* point for binary numbers.

For the decimal fraction 0.65625, the conversion is as follows:

	Fractional part	Integral part
0.65625 × 2	.31250	1
0.31250 × 2	.62500	0
0.62500 × 2	.25000	1
0.25000 × 2	.50000	0
0.50000 × 2	.00000	1

The binary fraction is 0.10101₂.

To convert a number like 71.375₁₀ to binary, we must convert the integral part by divisions and the fractional part by multiplication as illustrated below:

	Quotient	Remainder
71/2	35	1
35/2	17	1
17/2	8	1

	Quotient	Remainder
8/2	4	0
4/2	2	0
2/2	1	0
1/2	0	1

	Fractional part	Integral part
0.375 × 2	0.625	0
0.625 × 2	0.250	1
0.250 × 2	0.500	0
0.500 × 2	0.000	1

Therefore, decimal 71.375 is equivalent to 1 000 111. 010 1₂.

Of course, not all decimal fractions can be represented by a finite binary fraction. For example, the decimal fraction 0.4205 converts as follows:

	Fractional part	Integral part
0.4205 × 2	0.841	0
0.841 × 2	0.682	1
0.682 × 2	0.364	1
0.364 × 2	0.728	0
0.728 × 2	0.456	1
0.456 × 2	0.912	0
0.912 × 2	0.824	1
0.824 × 2	0.648	1
0.648 × 2	0.296	1
0.296 × 2	0.592	0
0.592 × 2	0.084	1
0.084 × 2	0.168	0

The binary fraction accurate to twelve digits is 0.0110 1011 1010₂. The decimal equivalent to six digits is 0.420360, as compared with the original 0.4205.

E-2. CONVERSION FROM DECIMAL TO OCTAL OR HEXADECIMAL

Conversion from decimal to octal or hexadecimal and vice versa is analogous to the conversion of decimal to binary and vice versa. For example, the conversion of 17022_d to octal is accomplished as follows:

	Quotient	Remainder
17022/8	2127	6
2127/8	265	7

	Quotient	Remainder
265/8	33	1
33/8	4	1
4/8	0	4

The octal equivalent to decimal 17022 is 41176₈.

Now for the useful property of octal numbers. Suppose we want to convert the octal number to a binary-coded octal number in much the same fashion that we converted a decimal number to binary-coded decimal. Since 7 is the largest digit encountered in octal, three binary digits are required to represent each octal digit. The conversion of 41176₈ to binary-coded octal is illustrated below:

Octal	4	1	1	7	6
Binary-coded octal	100	001	001	111	110

Recall that when we converted 17022_d to true binary, we obtained 100 001 001 111 110₂, which is identical to the binary-coded octal number derived above. In effect, when we take an octal number and convert digit by digit to binary, the result is identical to the true binary number. This stems from the fact that the base of the octal system, 8, is an integer power of 2.

In octal, fractions are treated in a similar fashion, with the digits being paired off in groups of three from the binary point:

10110111.01101 = 010 110 111 . 011 010 (base 2)

= 2 6 7 . 3 2 (base 8)

Zeros are added to the left or right so that an integral multiple of 3 is obtained.

The hexadecimal number system has similar characteristics. Conversion of 17022_d to hexadecimal is illustrated below:

	Quotient	Remainder
17022/16	1063	14 _d = E
1063/16	66	7
66/16	4	2
4/16	0	4

The hexadecimal number is 427E_h. Converting digit-by-digit to binary gives (four binary digits are needed to represent a hexadecimal digit)

4 2 7 E

0100 0010 0111 1110

which is equivalent to the true binary representation.

Conversion of binary fractions to hexadecimal is accomplished in a fashion analogous to the conversion of octal numbers, except that the binary digits are arranged in groups of four:

$$\begin{aligned} 10110111.01101 &= 1011\ 0111\ .\ 0110\ 1000\ (\text{base } 2) \\ &= \text{B}\quad 7\quad .\quad 6\quad 8\quad (\text{base } 16) \end{aligned}$$

591

Table 6-1 is quite helpful in making these conversions, but most experienced programmers are able to make such conversions without the aid of a table.

E-3. SUMMARY OF CONVERSIONS

Conversion between the binary, octal, decimal, and hexadecimal number systems is a task at which a student of computer science should be able to accomplish very readily. Table E-1 gives a complete set of conversions which should be noted carefully.

Table E-1. Conversion of Numbers	
Binary number:	10111010.10111
Conversion to octal:	010 111 010 . 101 110 ₂
	2 7 2 . 5 6 ₈
Conversion to decimal:	10111010.10111
	$= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5}$
	$= 128 + 0 + 32 + 16 + 8 + 0 + 2 + 0 + 0 + 0.5 + 0 + 0.125 + 0.0625 + 0.03125$
	$= 186.71875_d$
Conversion to hexadecimal:	1011 1010 . 1011 1000 ₂
	B A . B 8 _h
Conversion from octal to decimal:	272.56 ₈
	$= 2 \times 8^2 + 7 \times 8 + 2 + 5 \times 8^{-1} + 6 \times 8^{-2}$
	$= 128 + 56 + 2 + 0.125 + 0.09375$
	$= 186.71875_d$
Conversion from hexadecimal to decimal:	B A . B 8 _h
	$= 11 \times 16 + 10 + 11 \times 16^{-1} + 9 \times 16^{-2}$
	$= 176 + 10 + 0.6875 + 0.03125$
	$= 186.71875_d$
Conversion from decimal to octal:	

Table E-1. (Continued)

Quotient	Remainder	Fraction	Integral part
186/8 = 23	2	0.71875 × 8 = 0.75	5
23/8 = 2	7	0.75 × 8 = 0	6
2/8 = 0	2		

Octal number = 272.56₈

Conversion from decimal to hexadecimal:

Quotient	Remainder	Fraction	Integral part
186/16 = 11	10 = A _h	0.71875 × 16 = 0.5	11 = B ₁₆
11/16 = 0	11 = B _h	0.5 × 16 = 0	8

Hexadecimal number = BA.B8_h

Conversion from octal to hexadecimal and vice versa is rarely encountered but is easily accomplished via the binary intermediate.

E-4. INTRODUCTION TO BINARY ARITHMETIC

Arithmetic in the binary system is analogous to that in the decimal system. In fact, in some senses it is even simpler but unfortunately is not as familiar to us. For example, the addition table in the decimal system contains 100 entries, which most of us committed to memory at a very early age. In fact, we use it so frequently that we think nothing of it.

The addition table in the binary system is much simpler, consisting of only four entries as illustrated below:

+	0	1
0	0	1
1	1	10

Addition of 1 to 1 in binary produces a carry digit in the same fashion as adding 7 and 6 in the decimal system. As an example, consider following addition of two binary numbers:

1011	(carries)	1
10011		19
+ 1011		11
11110		30

The addition of 1 to 1 gives a 0 with a carry of 1; addition of a 1 to a 1 with a carry of 1 gives a 1 with another carry of 1.

Unfortunately, subtraction in this system gets out of hand owing to the borrowing problem. For example,

$$\begin{array}{r} 10 \\ 0\ 0\ 10\ \text{(borrowing)} \\ 11\cancel{1}\ \cancel{1}\ 0 \\ -10\ 1\ 1 \\ \hline 100\ 1\ 1 \end{array}$$

A more complex example is

$$\begin{array}{r} 10\ 1\ 1 \\ 0\ \cancel{1}\ 0\ \cancel{1}\ 0\ \cancel{1}\ 0\ \text{(borrowing)} \\ 1\ \cancel{1}\ 0\ 0\ 0\ 1 \\ -0\ 1\ 1\ 0\ 1\ 0 \\ \hline 0\ 1\ 0\ 1\ 1\ 1 \end{array}$$

With a little practice, one becomes quite good at it, but the process is nevertheless cumbersome.

Is there any way to subtract without borrowing? Consider the following decimal subtraction:

$$\begin{array}{r} 111014 \\ 5\ \cancel{1}\ 0\ \cancel{4}\ 14\ \text{(borrowing)} \\ \cancel{6}\ \cancel{2}\ \cancel{1}\ \cancel{5}\ \cancel{4} \\ 4\ 6\ 5\ 7\ 6 \\ \hline 1\ 5\ 5\ 7\ 8 \end{array}$$

Suppose we perform the process as follows:

$$62154 + (100000 - 46576) - 100000$$

This is equivalent to

$$62154 + 1 + (99999 - 46576) - 100000$$

The following subtraction can be performed without any borrowing:

$$\begin{array}{r} 99999 \\ -46576 \\ \hline 53423 \end{array}$$

594 Adding 1 gives 53434. Adding to 62154 gives

$$\begin{array}{r} 62154 \\ + 53424 \\ \hline 115578 \end{array}$$

Subtracting 100,000 gives the final result of 15,577.

The number 53,424 computed above is called the *tens complement* of -46,576. Essentially, it is obtained by subtracting 46,576 from 99,999 and adding 1 to the result.

In the binary system, a very similar procedure can be used. As we saw above, subtraction of 011010₂ from 110001₂ was somewhat involved owing to the borrowing. Suppose we compute the twos complement of 011010₂:

$$\begin{array}{r} 111111 \\ - 011010 \\ \hline 100101 \\ + 1 \\ \hline 100110 \end{array}$$

Note that the effect of the first subtraction was to change all the 1s in 011010 to 0s and all the 0s to 1. Therefore, the twos complement is obtained by simply changing all the 1s in the number to 0s, all the 0s to 1s, and adding 1 to the result.

To obtain the final answer, 100110 (the twos complement of 011010) is added to 110001 and 1000000 is subtracted from the result:

$$\begin{array}{r} 110001 \\ + 100110 \\ \hline 1010111 \\ - 1000000 \\ \hline 010111 \end{array}$$

No messy borrowing is encountered.

An alternate approach is to use a ones complement, where the number to be subtracted (the subtrahend) is first subtracted from 111111, which is equivalent to changing all 1s to 0s and 0s to 1s. This ones complement is added to the minuend, one is added to the result, and 1000000 subtracted to give the answer.

**E-5. ARITHMETIC IN THE OCTAL AND
HEXADECIMAL NUMBER SYSTEMS**

In reality, most programmers encounter the need to perform additions and subtractions of octal or hexadecimal numbers more frequently than for binary

numbers. Since we are familiar with decimal arithmetic, we shall first develop a procedure to add octal numbers using decimal arithmetic. Consider the following cases:

	Octal	Decimal
a.	3	3
	+2	+2
	5	5
b.	5	5
	+3	+3
	10	8
c.	5	5
	+6	+6
	13	11

Note that in case *a* decimal and octal arithmetic give the same answer. In cases *b* and *c* the answers are different because one of the following conditions exists:

- 1. The decimal addition produced one of the digits 8 or 9, which do not exist in the octal number system.
- 2. The decimal addition produced a carry.

In either case, the octal result can be obtained by simply adding 2 to the decimal result.

For example, consider adding the octal numbers 45173 and 16254. The process is carried out as follows:

45173
13254

58427
2 2

60447

(decimal addition)

(adjust)
(octal result)

The first step is to add the two numbers as if they were decimal. When the second digits from the right were added, a carry was produced, which necessitates the addition of 2 to this digit in the decimal result. The digit 8 in the decimal result is also removed by adding 2. With a little practice, one can mentally add 2 to the digits as necessary without showing explicitly a separate step.

Although subtraction of two octal digits can be accomplished with decimal arithmetic by subtracting 2 from certain digits, perhaps the most straightforward approach is to use the complement. For example, consider the subtraction of 14254 from 45173:

77777

-14254

63523

+1

63524

+45173

108697

222

130717

-100000

30717

(radix complement)

(decimal add)

(adjust)

(delete leading 1)

The first step is to obtain the radix-complement of the subtrahend by subtracting each digit from 7 and adding 1 to the result. Adding the complement to the minuend and deleting the leading 1 gives the desired answer.

Addition and subtraction of hexadecimal numbers can also be accomplished by adjusting the results of decimal arithmetic. For addition, the respective digits are added and units of 16 are removed for carries. For example, consider the addition of 42E52 and 3A5F2:

42E52

(14)

+3A5F2

(10)

77777

(12)

77777

(13)

77777

D

(15)

(20)

(19)

4

4

4

4

4

(decimal add)

(perform carries)

(express as hexadecimal)

The numbers enclosed in parentheses are the decimal equivalent to the respective hexadecimal digit. Again, with a little practice, this procedure can be executed without explicitly showing the extra steps.

As for octal, subtraction of hexadecimal numbers is readily accomplished using complements as illustrated for the subtraction of 3A572 from 42E52:

F

-3

C

(12)

C

F

A

5

5

F

5

A

(10)

A

F

7

8

8

F

2

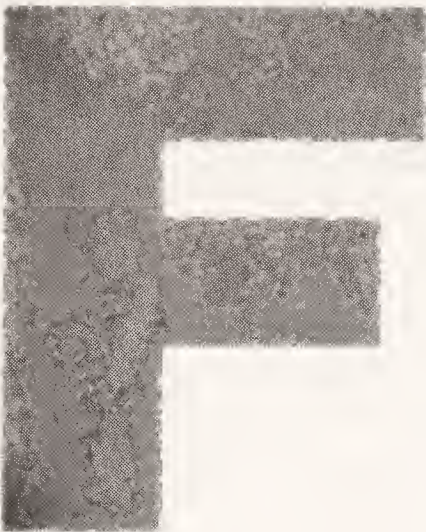
D

(14)

E

(radix-complement)

4	2	E	5	2	(decimal add)	597
(16)	7	(24)	(13)	(16)		
10	8	8	(14)	0	(perform carries)	
10	8	8	E	0	(express as hexadecimal)	
	8	8	E	0	(delete leading 1)	



CONVERSION TABLES

This appendix consists of four tables designed to facilitate the conversion of decimal to hexadecimal numbers and vice versa. Table F-1 gives the decimal equivalents of the powers of 2 and their reciprocals. Tables F-2 and F-3 are used to convert integer decimals to hexadecimal numbers and vice versa. Table F-4 is used for fractional numbers. Examples of the conversion of decimal numbers to hexadecimal follow:

Example 1 Convert decimal 2186 to hexadecimal. Table F-2 gives the answer 88A directly.

Example 2 Convert decimal 7,365,261 to hexadecimal. Table 2 gives direct conversions only for decimal numbers up to 4,096. For larger numbers, this table must be supplemented by the use of Table F-3. The steps are as follows:

- Step 1* Locate in Table F-3 the largest decimal number not exceeding 7,365,261. This is decimal 7,340,032, and corresponds to the hexadecimal number 700,000.
- Step 2* Subtract the decimal number found in the previous step from the original decimal number:

$$\begin{array}{r} 7,365,261 \\ -7,340,032 \\ \hline 25,229 \end{array}$$

- Step 3* This number is still larger than 4,096, so in essence Step 1 must be repeated with the new number. The largest entry in Table F-3 that does not exceed decimal 25,229 is decimal 24,576, which corresponds to hexadecimal 6,000.

Step 4 The two numbers are again subtracted as in Step 2 to produce a new number: 599

$$\begin{array}{r} 25,229 \\ 24,576 \\ \hline 653 \end{array}$$

Step 5 The hexadecimal equivalent of this number can be determined directly from Table F-2 to be 28D.

Step 6 The desired hexadecimal number is the sum of all hexadecimal numbers found in the previous steps:

$$\begin{array}{r} 700,000 \\ 6,000 \\ 28D \\ \hline 706,28D \end{array}$$

Example 3 Convert the fraction 0.343 to hexadecimal accurate to six hexadecimal digits. In essence the entries on the first page of Table F-4 give the first two hexadecimal digits, the entries on the second page of Table F-4 give the second two hexadecimal digits, etc. The steps are as follows:

Step 1 Locate the largest decimal entry from the first page of Table F-4 that does not exceed decimal 0.343. This is decimal 0.33984375 and corresponds to hexadecimal 0.570000. Subtract this number from 0.343:

$$\begin{array}{r} 0.34300000 \\ 0.33984375 \\ \hline 0.00315625 \end{array}$$

Step 2 Locate the largest decimal entry from the second page of Table F-4 that does not exceed decimal 0.00315625. This is decimal 0.0031433105 and corresponds to hexadecimal 0.00CE00. Again subtract the decimal numbers:

$$\begin{array}{r} 0.0031562500 \\ -0.0031433105 \\ \hline 0.0000129395 \end{array}$$

Step 3 Locate the decimal entry from the third page of Table F-4 that is nearest to decimal 0.0000129395. This is decimal 0.0000129342 and corresponds to hexadecimal 0.0000D9.

Step 4 Adding the hexadecimal numbers from previous steps gives the desired result:

0.570000
0.00CE00
0.0000D9

0.57CED9

Table F-1. Powers of Two

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.0625
32	5	0.0312 5
64	6	0.0156 25
128	7	0.0078 125
256	8	0.0039 0625
512	9	0.0019 5312 5
1024	10	0.0009 7656 25
2048	11	0.0004 8828 125
4096	12	0.0002 4414 0625
8192	13	0.0001 2207 0312 5
16384	14	0.0000 6103 5156 25
32768	15	0.0000 3051 7578 125
65536	16	0.0000 1525 8789 0625
131072	17	0.0000 0762 9394 5312 5
262144	18	0.0000 0381 4697 2656 25
524288	19	0.0000 0190 7348 6328 125
1048576	20	0.0000 0095 3674 3164 0625
2097152	21	0.0000 0047 6837 1582 0312 5
4194304	22	0.0000 0023 8418 5791 0156 25
8388608	23	0.0000 0011 9209 2895 5078 125
16777216	24	0.0000 0005 9604 6447 7539 0625
33554432	25	0.0000 0002 9802 3223 8769 5312 5
67108864	26	0.0000 0001 4901 1611 9384 7656 25
134217728	27	0.0000 0000 7450 5805 9692 3828 125
268435456	28	0.0000 0000 3725 2902 9846 1914 0625
536870912	29	0.0000 0000 1862 6451 4923 0957 0312 5
1073741824	30	0.0000 0000 0931 3225 7461 5478 5156 25
2147483648	31	0.0000 0000 0465 6612 8730 7739 2578 125
4294967296	32	0.0000 0000 0232 8306 4365 3869 6289 0625

Table F-2. Hexadecimal-Decimal Conversion Table for Decimal Numbers
Less than 4096

Highest-order hexadecimal digits	Lowest-order hexadecimal digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
01	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
02	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
03	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
04	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
05	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
06	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
07	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
08	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
09	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
0A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
0B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
0C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
0D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
0E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
0F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
10	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271
11	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
12	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303
13	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319
14	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335
15	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351
16	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367
17	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383
18	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399
19	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415
1A	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431
1B	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447
1C	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463
1D	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479
1E	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495
1F	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511
20	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527
21	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543
22	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559
23	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575
24	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591
25	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607
26	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623
27	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639
28	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655
29	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671
2A	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687
2B	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703
2C	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719
2D	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735
2E	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751
2F	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767
30	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783
31	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799
32	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815
33	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831
34	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847
35	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863
36	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879
37	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895
38	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911
39	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927
3A	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943
3B	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959
3C	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975
3D	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991
3E	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007
3F	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

Table F-2. (Continued)

Highest-order hexadecimal digits		Lowest-order hexadecimal digit															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	4C	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
	41	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
	42	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
	43	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
	44	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
	45	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
	46	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
	47	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
	48	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
	49	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
	4A	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
	4B	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
	4C	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
	4D	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
	4E	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
	4F	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
	50	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
	51	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
	52	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
	53	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
	54	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
	55	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
	56	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
	57	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
	58	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
	59	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
	5A	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
	5B	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
	5C	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
	5D	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
	5E	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
	5F	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
	60	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
	61	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
	62	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
	63	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
	64	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
	65	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
	66	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
	67	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
	68	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
	69	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
	6A	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
	6B	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
	6C	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
	6D	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
	6E	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
	6F	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
	70	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
	71	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
	72	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
	73	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
	74	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
	75	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
	76	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
	77	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
	78	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
	79	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
	7A	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
	7B	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
	7C	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
	7D	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
	7E	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
	7F	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

Table F-2. (Continued)

Highest-order hexadecimal digits	Lowest-order hexadecimal digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	2048	2049	205C	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

Table F-2. (Continued)

Highest-order hexadecimal digits	Lowest-order hexadecimal digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Table F-3. Hexadecimal-Decimal Conversion Table for Large Decimal Numbers

HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL
1 000	4 096	10 0000	1 048 576	1000 0000	268 435 456
2 000	8 192	20 0000	2 097 152	2000 0000	536 870 912
3 000	12 288	30 0000	3 145 728	3000 0000	805 306 368
4 000	16 384	40 0000	4 194 304	4000 0000	1 073 741 824
5 000	20 480	50 0000	5 242 880	5000 0000	1 342 177 280
6 000	24 576	60 0000	6 291 456	6000 0000	1 610 612 736
7 000	28 672	70 0000	7 340 032	7000 0000	1 879 048 192
8 000	32 768	80 0000	8 388 608	8000 0000	2 147 483 648
9 000	36 864	90 0000	9 437 184	9000 0000	2 415 919 104
A 000	40 960	A0 0000	10 485 760	A000 0000	2 684 354 560
B 000	45 056	B0 0000	11 534 336	B000 0000	2 952 790 016
C 000	49 152	C0 0000	12 582 912	C000 0000	3 221 225 472
D 000	53 248	D0 0000	13 631 488	D000 0000	3 489 660 928
E 000	57 344	E0 0000	14 680 064	E000 0000	3 758 096 384
F 000	61 440	F0 0000	15 728 640	F000 0000	4 026 531 840
1 0000	65 536	100 0000	16 777 216	1 0000 0000	4 294 967 296
2 0000	131 072	200 0000	33 554 432	2 0000 0000	8 589 934 592
3 0000	196 608	300 0000	50 331 648	3 0000 0000	12 884 901 888
4 0000	262 144	400 0000	67 108 864	4 0000 0000	17 179 869 184
5 0000	327 680	500 0000	83 886 080	5 0000 0000	21 474 836 480
6 0000	393 216	600 0000	100 663 296	6 0000 0000	25 769 803 776
7 0000	458 752	700 0000	117 440 512	7 0000 0000	30 064 771 072
8 0000	524 288	800 0000	134 217 728	8 0000 0000	34 359 738 368
9 0000	589 824	900 0000	150 994 944	9 0000 0000	38 654 705 664
A 0000	655 360	A00 0000	167 772 160	A 0000 0000	42 949 672 960
B 0000	720 896	B00 0000	184 549 376	B 0000 0000	47 244 640 256
C 0000	786 432	C00 0000	201 326 592	C 0000 0000	51 539 607 552
D 0000	851 968	D00 0000	218 103 808	D 0000 0000	55 834 574 848
E 0000	917 504	E00 0000	234 881 024	E 0000 0000	60 129 542 144
F 0000	983 040	F00 0000	251 658 240	F 0000 0000	64 424 509 440

Table F-4. Conversion Table for Fractional Numbers

HEXADECIMAL		DECIMAL		HEXADECIMAL		DECIMAL	
0.0C00	0000	0.000C0	00000	0.4000	0000	0.25000	00000
0.0100	0000	0.00390	62500	0.4100	0000	0.25390	62500
0.0200	0000	0.00781	25000	0.4200	0000	0.25781	25000
0.0300	0000	0.01171	87500	0.4300	0000	0.26171	87500
0.0400	0000	0.01562	50000	0.4400	0000	0.26562	50000
0.0500	0000	0.01953	12500	0.4500	0000	0.26953	12500
0.0600	0000	0.02343	75000	0.4600	0000	0.27343	75000
0.0700	0000	0.02734	37500	0.4700	0000	0.27734	37500
0.0800	0000	0.03125	00000	0.4800	0000	0.28125	00000
0.0900	0000	0.03515	62500	0.4900	0000	0.28515	62500
0.0A00	0000	0.03906	25000	0.4A00	0000	0.28906	25000
0.0B00	0000	0.04296	87500	0.4B00	0000	0.29296	87500
0.0C00	0000	0.04687	50000	0.4C00	0000	0.29687	50000
0.0D00	0000	0.05078	12500	0.4D00	0000	0.30078	12500
0.0E00	0000	0.05468	75000	0.4E00	0000	0.30468	75000
0.0F00	0000	0.05859	37500	0.4F00	0000	0.30859	37500
0.1000	0000	0.06250	00000	0.5000	0000	0.31250	00000
0.1100	0000	0.06640	62500	0.5100	0000	0.31640	62500
0.1200	0000	0.07031	25000	0.5200	0000	0.32031	25000
0.1300	0000	0.07421	87500	0.5300	0000	0.32421	87500
0.1400	0000	0.07812	50000	0.5400	0000	0.32812	50000
0.1500	0000	0.08203	12500	0.5500	0000	0.33203	12500
0.1600	0000	0.08593	75000	0.5600	0000	0.33593	75000
0.1700	0000	0.08984	37500	0.5700	0000	0.33984	37500
0.1800	0000	0.09375	00000	0.5800	0000	0.34375	00000
0.1900	0000	0.09765	62500	0.5900	0000	0.34765	62500
0.1A00	0000	0.10156	25000	0.5A00	0000	0.35156	25000
0.1B00	0000	0.10546	87500	0.5B00	0000	0.35546	87500
0.1C00	0000	0.10937	50000	0.5C00	0000	0.35937	50000
0.1D00	0000	0.11328	12500	0.5D00	0000	0.36328	12500
0.1E00	0000	0.11718	75000	0.5E00	0000	0.36718	75000
0.1F00	0000	0.12109	37500	0.5F00	0000	0.37109	37500
0.2000	0000	0.12500	00000	0.6000	0000	0.37500	00000
0.2100	0000	0.12890	62500	0.6100	0000	0.37890	62500
0.2200	0000	0.13281	25000	0.6200	0000	0.38281	25000
0.2300	0000	0.13671	87500	0.6300	0000	0.38671	87500
0.2400	0000	0.14062	50000	0.6400	0000	0.39062	50000
0.2500	0000	0.14453	12500	0.6500	0000	0.39453	12500
0.2600	0000	0.14843	75000	0.6600	0000	0.39843	75000
0.2700	0000	0.15234	37500	0.6700	0000	0.40234	37500
0.2800	0000	0.15625	00000	0.6800	0000	0.40625	00000
0.2900	0000	0.16015	62500	0.6900	0000	0.41015	62500
0.2A00	0000	0.16406	25000	0.6A00	0000	0.41406	25000
0.2B00	0000	0.16796	87500	0.6B00	0000	0.41796	87500
0.2C00	0000	0.17187	50000	0.6C00	0000	0.42187	50000
0.2D00	0000	0.17578	12500	0.6D00	0000	0.42578	12500
0.2E00	0000	0.17968	75000	0.6E00	0000	0.42968	75000
0.2F00	0000	0.18359	37500	0.6F00	0000	0.43359	37500
0.3000	0000	0.18750	00000	0.7000	0000	0.43750	00000
0.3100	0000	0.19140	62500	0.7100	0000	0.44140	62500
0.3200	0000	0.19531	25000	0.7200	0000	0.44531	25000
0.3300	0000	0.19921	87500	0.7300	0000	0.44921	87500
0.3400	0000	0.20312	50000	0.7400	0000	0.45312	50000
0.3500	0000	0.20703	12500	0.7500	0000	0.45703	12500
0.3600	0000	0.21093	75000	0.7600	0000	0.46093	75000
0.3700	0000	0.21484	37500	0.7700	0000	0.46484	37500
0.3800	0000	0.21875	00000	0.7800	0000	0.46875	00000
0.3900	0000	0.22265	62500	0.7900	0000	0.47265	62500
0.3A00	0000	0.22656	25000	0.7A00	0000	0.47656	25000
0.3B00	0000	0.23046	87500	0.7B00	0000	0.48046	87500
0.3C00	0000	0.23437	50000	0.7C00	0000	0.48437	50000
0.3D00	0000	0.23828	12500	0.7D00	0000	0.48828	12500
0.3E00	0000	0.24218	75000	0.7E00	0000	0.49218	75000
0.3F00	0000	0.24609	37500	0.7F00	0000	0.49609	37500

Table F-4. (Continued)

HEXADECIMAL		DECIMAL		HEXADECIMAL		DECIMAL	
C.8000	0000	C.50000	00000	C.C000	0000	C.75000	00000
C.8100	0000	C.50390	62500	C.C100	0000	C.75390	62500
C.8200	0000	C.50781	25000	C.C200	0000	C.75781	25000
C.8300	0000	C.51171	87500	C.C300	0000	C.76171	87500
C.8400	0000	C.51562	50000	C.C400	0000	C.76562	50000
C.8500	0000	C.51953	12500	C.C500	0000	C.76953	12500
C.8600	0000	C.52343	75000	C.C600	0000	C.77343	75000
C.8700	0000	C.52734	37500	C.C700	0000	C.77734	37500
C.8800	0000	C.53125	00000	C.C800	0000	C.78125	00000
C.8900	0000	C.53515	62500	C.C900	0000	C.78515	62500
C.8A00	0000	C.53906	25000	C.CA00	0000	C.78906	25000
C.8B00	0000	C.54296	87500	C.CB00	0000	C.79296	87500
C.8C00	0000	C.54687	50000	C.CC00	0000	C.79687	50000
C.8D00	0000	C.55078	12500	C.CD00	0000	C.80078	12500
C.8E00	0000	C.55468	75000	C.CE00	0000	C.80468	75000
C.8F00	0000	C.55859	37500	C.CF00	0000	C.80859	37500
C.9000	0000	C.56250	00000	C.D000	0000	C.81250	00000
C.9100	0000	C.56640	62500	C.D100	0000	C.81640	62500
C.9200	0000	C.57031	25000	C.D200	0000	C.82031	25000
C.9300	0000	C.57421	87500	C.D300	0000	C.82421	87500
C.9400	0000	C.57812	50000	C.D400	0000	C.82812	50000
C.9500	0000	C.58203	12500	C.D500	0000	C.83203	12500
C.9600	0000	C.58593	75000	C.D600	0000	C.83593	75000
C.9700	0000	C.58984	37500	C.D700	0000	C.83984	37500
C.9800	0000	C.59375	00000	C.D800	0000	C.84375	00000
C.9900	0000	C.59765	62500	C.D900	0000	C.84765	62500
C.9A00	0000	C.60156	25000	C.DA00	0000	C.85156	25000
C.9B00	0000	C.60546	87500	C.DB00	0000	C.85546	87500
C.9C00	0000	C.60937	50000	C.DC00	0000	C.85937	50000
C.9D00	0000	C.61328	12500	C.DD00	0000	C.86328	12500
C.9E00	0000	C.61718	75000	C.DE00	0000	C.86718	75000
C.9F00	0000	C.62109	37500	C.DF00	0000	C.87109	37500
C.A000	0000	C.62500	00000	C.E000	0000	C.87500	00000
C.A100	0000	C.62890	62500	C.E100	0000	C.87890	62500
C.A200	0000	C.63281	25000	C.E200	0000	C.88281	25000
C.A300	0000	C.63671	87500	C.E300	0000	C.88671	87500
C.A400	0000	C.64062	50000	C.E400	0000	C.89062	50000
C.A500	0000	C.64453	12500	C.E500	0000	C.89453	12500
C.A600	0000	C.64843	75000	C.E600	0000	C.89843	75000
C.A700	0000	C.65234	37500	C.E700	0000	C.90234	37500
C.A800	0000	C.65625	00000	C.E800	0000	C.90625	00000
C.A900	0000	C.66015	62500	C.E900	0000	C.91015	62500
C.AA00	0000	C.66406	25000	C.EA00	0000	C.91406	25000
C.AB00	0000	C.66796	87500	C.EB00	0000	C.91796	87500
C.AC00	0000	C.67187	50000	C.EC00	0000	C.92187	50000
C.AD00	0000	C.67578	12500	C.ED00	0000	C.92578	12500
C.AE00	0000	C.67968	75000	C.EE00	0000	C.92968	75000
C.AF00	0000	C.68359	37500	C.EF00	0000	C.93359	37500
C.B000	0000	C.68750	00000	C.F000	0000	C.93750	00000
C.B100	0000	C.69140	62500	C.F100	0000	C.94140	62500
C.B200	0000	C.69531	25000	C.F200	0000	C.94531	25000
C.B300	0000	C.69921	87500	C.F300	0000	C.94921	87500
C.B400	0000	C.70312	50000	C.F400	0000	C.95312	50000
C.B500	0000	C.70703	12500	C.F500	0000	C.95703	12500
C.B600	0000	C.71093	75000	C.F600	0000	C.96093	75000
C.B700	0000	C.71484	37500	C.F700	0000	C.96484	37500
C.B800	0000	C.71875	00000	C.F800	0000	C.96875	00000
C.B900	0000	C.72265	62500	C.F900	0000	C.97265	62500
C.BA00	0000	C.72656	25000	C.FA00	0000	C.97656	25000
C.BB00	0000	C.73046	87500	C.FB00	0000	C.98046	87500
C.BC00	0000	C.73437	50000	C.FC00	0000	C.98437	50000
C.BD00	0000	C.73828	12500	C.FD00	0000	C.98828	12500
C.BE00	0000	C.74218	75000	C.FE00	0000	C.99218	75000
C.BF00	0000	C.74609	37500	C.FF00	0000	C.99609	37500

Table F-4. (Continued)

HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL
0.0000 0000	0.00000 00000	0.00C40 0000	0.00097 65625
0.0001 0000	0.00001 52588	0.00041 0000	0.00099 18213
0.0002 0000	0.00003 05176	0.00042 0000	0.00100 70801
0.0003 0000	0.00004 57764	0.00043 0000	0.00102 23389
0.0004 0000	0.00006 10352	0.00044 0000	0.00103 75977
0.0005 0000	0.00007 62939	0.00045 0000	0.00105 28564
0.0006 0000	0.00009 15527	0.00046 0000	0.00106 81152
0.0007 0000	0.00010 68115	0.00047 0000	0.00108 33740
0.0008 0000	0.00012 20703	0.00048 0000	0.00109 86328
0.0009 0000	0.00013 73291	0.00049 0000	0.00111 38916
0.000A 0000	0.00015 25879	0.0004A 0000	0.00112 91504
0.000B 0000	0.00016 78467	0.0004B 0000	0.00114 44092
0.000C 0000	0.00018 31055	0.0004C 0000	0.00115 96680
0.000D 0000	0.00019 83643	0.0004D 0000	0.00117 49268
0.000E 0000	0.00021 36230	0.0004E 0000	0.00119 01855
0.000F 0000	0.00022 88818	0.0004F 0000	0.00120 54443
0.0010 0000	0.00024 41406	0.00050 0000	0.00122 07031
0.0011 0000	0.00025 93994	0.00051 0000	0.00123 59619
0.0012 0000	0.00027 46582	0.00052 0000	0.00125 12207
0.0013 0000	0.00028 99170	0.00053 0000	0.00126 64795
0.0014 0000	0.00030 51758	0.00054 0000	0.00128 17383
0.0015 0000	0.00032 04346	0.00055 0000	0.00129 69971
0.0016 0000	0.00033 56934	0.00056 0000	0.00131 22559
0.0017 0000	0.00035 09521	0.00057 0000	0.00132 75146
0.0018 0000	0.00036 62109	0.00058 0000	0.00134 27734
0.0019 0000	0.00038 14697	0.00059 0000	0.00135 80322
0.001A 0000	0.00039 67285	0.0005A 0000	0.00137 32910
0.001B 0000	0.00041 19873	0.0005B 0000	0.00138 85498
0.001C 0000	0.00042 72461	0.0005C 0000	0.00140 38086
0.001D 0000	0.00044 25049	0.0005D 0000	0.00141 90674
0.001E 0000	0.00045 77637	0.0005E 0000	0.00143 43262
0.001F 0000	0.00047 30225	0.0005F 0000	0.00144 95850
0.0020 0000	0.00048 82813	0.00060 0000	0.00146 48438
0.0021 0000	0.00050 35400	0.00061 0000	0.00148 01025
0.0022 0000	0.00051 87988	0.00062 0000	0.00149 53613
0.0023 0000	0.00053 40576	0.00063 0000	0.00151 06201
0.0024 0000	0.00054 93164	0.00064 0000	0.00152 58789
0.0025 0000	0.00056 45752	0.00065 0000	0.00154 11377
0.0026 0000	0.00057 98340	0.00066 0000	0.00155 63965
0.0027 0000	0.00059 50928	0.00067 0000	0.00157 16553
0.0028 0000	0.00061 03516	0.00068 0000	0.00158 69141
0.0029 0000	0.00062 56104	0.00069 0000	0.00160 21729
0.002A 0000	0.00064 08691	0.0006A 0000	0.00161 74316
0.002B 0000	0.00065 61279	0.0006B 0000	0.00163 26904
0.002C 0000	0.00067 13867	0.0006C 0000	0.00164 79492
0.002D 0000	0.00068 66455	0.0006D 0000	0.00166 32080
0.002E 0000	0.00070 19043	0.0006E 0000	0.00167 84668
0.002F 0000	0.00071 71631	0.0006F 0000	0.00169 37256
0.0030 0000	0.00073 24219	0.00070 0000	0.00170 89844
0.0031 0000	0.00074 76807	0.00071 0000	0.00172 42432
0.0032 0000	0.00076 29395	0.00072 0000	0.00173 95020
0.0033 0000	0.00077 81982	0.00073 0000	0.00175 47607
0.0034 0000	0.00079 34570	0.00074 0000	0.00177 00195
0.0035 0000	0.00080 87158	0.00075 0000	0.00178 52783
0.0036 0000	0.00082 39746	0.00076 0000	0.00180 05371
0.0037 0000	0.00083 92334	0.00077 0000	0.00181 57959
0.0038 0000	0.00085 44922	0.00078 0000	0.00183 10547
0.0039 0000	0.00086 97510	0.00079 0000	0.00184 63135
0.003A 0000	0.00088 50098	0.0007A 0000	0.00186 15723
0.003B 0000	0.00090 02686	0.0007B 0000	0.00187 68311
0.003C 0000	0.00091 55273	0.0007C 0000	0.00189 20898
0.003D 0000	0.00093 07861	0.0007D 0000	0.00190 73486
0.003E 0000	0.00094 60449	0.0007E 0000	0.00192 26074
0.003F 0000	0.00096 13037	0.0007F 0000	0.00193 78662

Table F-4. (Continued)

HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL
0.0080 0000	0.00195 31250	0.00C0 0000	0.00292 96875
0.0081 0000	0.00196 83838	0.00C1 0000	0.00294 49463
0.0082 0000	0.00198 36426	0.00C2 0000	0.00296 02051
0.0083 0000	0.00199 89014	0.00C3 0000	0.00297 54639
0.0084 0000	0.00201 41602	0.00C4 0000	0.00299 07227
0.0085 0000	0.00202 94189	0.00C5 0000	0.00300 59814
0.0086 0000	0.00204 46777	0.00C6 0000	0.00302 12402
0.0087 0000	0.00205 99365	0.00C7 0000	0.00303 64990
0.0088 0000	0.00207 51953	0.00C8 0000	0.00305 17578
0.0089 0000	0.00209 04541	0.00C9 0000	0.00306 70166
0.008A 0000	0.00210 57129	0.00CA 0000	0.00308 22754
0.008B 0000	0.00212 09717	0.00CB 0000	0.00309 75342
0.008C 0000	0.00213 62305	0.00CC 0000	0.00311 27930
0.008D 0000	0.00215 14893	0.00CD 0000	0.00312 80518
0.008E 0000	0.00216 67480	0.00CE 0000	0.00314 33105
0.008F 0000	0.00218 20068	0.00CF 0000	0.00315 85693
0.0090 0000	0.00219 72656	0.00D0 0000	0.00317 38281
0.0091 0000	0.00221 25244	0.00D1 0000	0.00318 90869
0.0092 0000	0.00222 77832	0.00D2 0000	0.00320 43457
0.0093 0000	0.00224 30420	0.00D3 0000	0.00321 96045
0.0094 0000	0.00225 83008	0.00D4 0000	0.00323 48633
0.0095 0000	0.00227 35596	0.00D5 0000	0.00325 01221
0.0096 0000	0.00228 88184	0.00D6 0000	0.00326 53809
0.0097 0000	0.00230 40771	0.00D7 0000	0.00328 06396
0.0098 0000	0.00231 93359	0.00D8 0000	0.00329 58984
0.0099 0000	0.00233 45947	0.00D9 0000	0.00331 11572
0.009A 0000	0.00234 98535	0.00DA 0000	0.00332 64160
0.009B 0000	0.00236 51123	0.00DB 0000	0.00334 16748
0.009C 0000	0.00238 03711	0.00DC 0000	0.00335 69336
0.009D 0000	0.00239 56299	0.00DD 0000	0.00337 21924
0.009E 0000	0.00241 08887	0.00DE 0000	0.00338 74512
0.009F 0000	0.00242 61475	0.00DF 0000	0.00340 27100
0.00A0 0000	0.00244 14063	0.00E0 0000	0.00341 79688
0.00A1 0000	0.00245 66650	0.00E1 0000	0.00343 32275
0.00A2 0000	0.00247 19238	0.00E2 0000	0.00344 84863
0.00A3 0000	0.00248 71826	0.00E3 0000	0.00346 37451
0.00A4 0000	0.00250 24414	0.00E4 0000	0.00347 90039
0.00A5 0000	0.00251 77002	0.00E5 0000	0.00349 42627
0.00A6 0000	0.00253 29590	0.00E6 0000	0.00350 95215
0.00A7 0000	0.00254 82178	0.00E7 0000	0.00352 47803
0.00A8 0000	0.00256 34766	0.00E8 0000	0.00354 00391
0.00A9 0000	0.00257 87354	0.00E9 0000	0.00355 52979
0.00AA 0000	0.00259 39941	0.00EA 0000	0.00357 05566
0.00AB 0000	0.00260 92529	0.00EB 0000	0.00358 58154
0.00AC 0000	0.00262 45117	0.00EC 0000	0.00360 10742
0.00AD 0000	0.00263 97705	0.00ED 0000	0.00361 63330
0.00AE 0000	0.00265 50293	0.00EE 0000	0.00363 15918
0.00AF 0000	0.00267 02881	0.00EF 0000	0.00364 68506
0.00B0 0000	0.00268 55469	0.00F0 0000	0.00366 21094
0.00B1 0000	0.00270 08057	0.00F1 0000	0.00367 73682
0.00B2 0000	0.00271 60645	0.00F2 0000	0.00369 26270
0.00B3 0000	0.00273 13232	0.00F3 0000	0.00370 78857
0.00B4 0000	0.00274 65820	0.00F4 0000	0.00372 31445
0.00B5 0000	0.00276 18408	0.00F5 0000	0.00373 84033
0.00B6 0000	0.00277 70996	0.00F6 0000	0.00375 36621
0.00B7 0000	0.00279 23584	0.00F7 0000	0.00376 89209
0.00B8 0000	0.00280 76172	0.00F8 0000	0.00378 41797
0.00B9 0000	0.00282 28760	0.00F9 0000	0.00379 94385
0.00BA 0000	0.00283 81348	0.00FA 0000	0.00381 46973
0.00BB 0000	0.00285 33936	0.00FB 0000	0.00382 99561
0.00BC 0000	0.00286 86523	0.00FC 0000	0.00384 52148
0.00BD 0000	0.00288 39111	0.00FD 0000	0.00386 04736
0.00BE 0000	0.00289 91699	0.00FE 0000	0.00387 57324
0.00BF 0000	0.00291 44287	0.00FF 0000	0.00389 09912

Table F-4. (Continued)

HEXADECIMAL		DECIMAL		HEXADECIMAL		DECIMAL	
0.0000	0000	0.00000	00000	0.0000	4000	0.00000	38147
0.0000	0100	0.00000	00596	0.0000	4100	0.00000	38743
0.0000	0200	0.00000	01192	0.0000	4200	0.00000	39339
0.0000	0300	0.00000	01788	0.0000	4300	0.00000	39935
0.0000	0400	0.00000	02384	0.0000	4400	0.00000	40531
0.0000	0500	0.00000	02980	0.0000	4500	0.00000	41127
0.0000	0600	0.00000	03576	0.0000	4600	0.00000	41723
0.0000	0700	0.00000	04172	0.0000	4700	0.00000	42319
0.0000	0800	0.00000	04768	0.0000	4800	0.00000	42915
0.0000	0900	0.00000	05364	0.0000	4900	0.00000	43511
0.0000	0A00	0.00000	05960	0.0000	4A00	0.00000	44107
0.0000	0B00	0.00000	06557	0.0000	4B00	0.00000	44703
0.0000	0C00	0.00000	07153	0.0000	4C00	0.00000	45300
0.0000	0D00	0.00000	07749	0.0000	4D00	0.00000	45896
0.0000	0E00	0.00000	08345	0.0000	4E00	0.00000	46492
0.0000	0F00	0.00000	08941	0.0000	4F00	0.00000	47088
0.0000	1000	0.00000	09537	0.0000	5000	0.00000	47684
0.0000	1100	0.00000	10133	0.0000	5100	0.00000	48280
0.0000	1200	0.00000	10729	0.0000	5200	0.00000	48876
0.0000	1300	0.00000	11325	0.0000	5300	0.00000	49472
0.0000	1400	0.00000	11921	0.0000	5400	0.00000	50068
0.0000	1500	0.00000	12517	0.0000	5500	0.00000	50664
0.0000	1600	0.00000	13113	0.0000	5600	0.00000	51260
0.0000	1700	0.00000	13709	0.0000	5700	0.00000	51856
0.0000	1800	0.00000	14305	0.0000	5800	0.00000	52452
0.0000	1900	0.00000	14901	0.0000	5900	0.00000	53048
0.0000	1A00	0.00000	15497	0.0000	5A00	0.00000	53644
0.0000	1B00	0.00000	16093	0.0000	5B00	0.00000	54240
0.0000	1C00	0.00000	16689	0.0000	5C00	0.00000	54836
0.0000	1D00	0.00000	17285	0.0000	5D00	0.00000	55432
0.0000	1E00	0.00000	17881	0.0000	5E00	0.00000	56028
0.0000	1F00	0.00000	18477	0.0000	5F00	0.00000	56624
0.0000	2000	0.00000	19073	0.0000	6000	0.00000	57220
0.0000	2100	0.00000	19670	0.0000	6100	0.00000	57817
0.0000	2200	0.00000	20266	0.0000	6200	0.00000	58413
0.0000	2300	0.00000	20862	0.0000	6300	0.00000	59009
0.0000	2400	0.00000	21458	0.0000	6400	0.00000	59605
0.0000	2500	0.00000	22054	0.0000	6500	0.00000	60201
0.0000	2600	0.00000	22650	0.0000	6600	0.00000	60797
0.0000	2700	0.00000	23246	0.0000	6700	0.00000	61393
0.0000	2800	0.00000	23842	0.0000	6800	0.00000	61989
0.0000	2900	0.00000	24438	0.0000	6900	0.00000	62585
0.0000	2A00	0.00000	25034	0.0000	6A00	0.00000	63181
0.0000	2B00	0.00000	25630	0.0000	6B00	0.00000	63777
0.0000	2C00	0.00000	26226	0.0000	6C00	0.00000	64373
0.0000	2D00	0.00000	26822	0.0000	6D00	0.00000	64969
0.0000	2E00	0.00000	27418	0.0000	6E00	0.00000	65565
0.0000	2F00	0.00000	28014	0.0000	6F00	0.00000	66161
0.0000	3000	0.00000	28610	0.0000	7000	0.00000	66757
0.0000	3100	0.00000	29206	0.0000	7100	0.00000	67353
0.0000	3200	0.00000	29802	0.0000	7200	0.00000	67949
0.0000	3300	0.00000	30398	0.0000	7300	0.00000	68545
0.0000	3400	0.00000	30994	0.0000	7400	0.00000	69141
0.0000	3500	0.00000	31590	0.0000	7500	0.00000	69737
0.0000	3600	0.00000	32187	0.0000	7600	0.00000	70333
0.0000	3700	0.00000	32783	0.0000	7700	0.00000	70930
0.0000	3800	0.00000	33379	0.0000	7800	0.00000	71526
0.0000	3900	0.00000	33975	0.0000	7900	0.00000	72122
0.0000	3A00	0.00000	34571	0.0000	7A00	0.00000	72718
0.0000	3B00	0.00000	35167	0.0000	7B00	0.00000	73314
0.0000	3C00	0.00000	35763	0.0000	7C00	0.00000	73910
0.0000	3D00	0.00000	36359	0.0000	7D00	0.00000	74506
0.0000	3E00	0.00000	36955	0.0000	7E00	0.00000	75102
0.0000	3F00	0.00000	37551	0.0000	7F00	0.00000	75698

Table F-4. (Continued)

HEXADECIMAL		DECIMAL		HEXADECIMAL		DECIMAL	
0.0000	8000	0.00000	76294	0.0000	C000	0.00001	14441
0.0000	8100	0.00000	76890	0.0000	C100	0.00001	15037
0.0000	8200	0.00000	77486	0.0000	C200	0.00001	15633
0.0000	8300	0.00000	78082	0.0000	C300	0.00001	16229
0.0000	8400	0.00000	78678	0.0000	C400	0.00001	16825
0.0000	8500	0.00000	79274	0.0000	C500	0.00001	17421
0.0000	8600	0.00000	79870	0.0000	C600	0.00001	18017
0.0000	8700	0.00000	80466	0.0000	C700	0.00001	18613
0.0000	8800	0.00000	81062	0.0000	C800	0.00001	19209
0.0000	8900	0.00000	81658	0.0000	C900	0.00001	19805
0.0000	8A00	0.00000	82254	0.0000	CA00	0.00001	20401
0.0000	8B00	0.00000	82850	0.0000	CB00	0.00001	20997
0.0000	8C00	0.00000	83447	0.0000	CC00	0.00001	21593
0.0000	8D00	0.00000	84043	0.0000	CD00	0.00001	22190
0.0000	8E00	0.00000	84639	0.0000	CE00	0.00001	22786
0.0000	8F00	0.00000	85235	0.0000	CF00	0.00001	23382
0.0000	9000	0.00000	85831	0.0000	D000	0.00001	23978
0.0000	9100	0.00000	86427	0.0000	D100	0.00001	24574
0.0000	9200	0.00000	87023	0.0000	D200	0.00001	25170
0.0000	9300	0.00000	87619	0.0000	D300	0.00001	25766
0.0000	9400	0.00000	88215	0.0000	D400	0.00001	26362
0.0000	9500	0.00000	88811	0.0000	D500	0.00001	26958
0.0000	9600	0.00000	89407	0.0000	D600	0.00001	27554
0.0000	9700	0.00000	90003	0.0000	D700	0.00001	28150
0.0000	9800	0.00000	90599	0.0000	D800	0.00001	28746
0.0000	9900	0.00000	91195	0.0000	D900	0.00001	29342
0.0000	9A00	0.00000	91791	0.0000	DA00	0.00001	29938
0.0000	9B00	0.00000	92387	0.0000	DB00	0.00001	30534
0.0000	9C00	0.00000	92983	0.0000	DC00	0.00001	31130
0.0000	9D00	0.00000	93579	0.0000	DD00	0.00001	31726
0.0000	9E00	0.00000	94175	0.0000	DE00	0.00001	32322
0.0000	9F00	0.00000	94771	0.0000	DF00	0.00001	32918
0.0000	A000	0.00000	95367	0.0000	E000	0.00001	33514
0.0000	A100	0.00000	95963	0.0000	E100	0.00001	34110
0.0000	A200	0.00000	96560	0.0000	E200	0.00001	34706
0.0000	A300	0.00000	97156	0.0000	E300	0.00001	35303
0.0000	A400	0.00000	97752	0.0000	E400	0.00001	35899
0.0000	A500	0.00000	98348	0.0000	E500	0.00001	36495
0.0000	A600	0.00000	98944	0.0000	E600	0.00001	37091
0.0000	A700	0.00000	99540	0.0000	E700	0.00001	37687
0.0000	A800	0.00001	00136	0.0000	E800	0.00001	38283
0.0000	A900	0.00001	00732	0.0000	E900	0.00001	38879
0.0000	AA00	0.00001	01328	0.0000	EA00	0.00001	39475
0.0000	AB00	0.00001	01924	0.0000	EB00	0.00001	40071
0.0000	AC00	0.00001	02520	0.0000	EC00	0.00001	40667
0.0000	AD00	0.00001	03116	0.0000	ED00	0.00001	41263
0.0000	AE00	0.00001	03712	0.0000	EE00	0.00001	41859
0.0000	AF00	0.00001	04308	0.0000	EF00	0.00001	42455
0.0000	B000	0.00001	04904	0.0000	FG00	0.00001	43051
0.0000	B100	0.00001	05500	0.0000	F100	0.00001	43647
0.0000	B200	0.00001	06096	0.0000	F200	0.00001	44243
0.0000	B300	0.00001	06692	0.0000	F300	0.00001	44839
0.0000	B400	0.00001	07288	0.0000	F400	0.00001	45435
0.0000	B500	0.00001	07884	0.0000	F500	0.00001	46031
0.0000	B600	0.00001	08480	0.0000	F600	0.00001	46627
0.0000	B700	0.00001	09076	0.0000	F700	0.00001	47223
0.0000	B800	0.00001	09673	0.0000	F800	0.00001	47820
0.0000	B900	0.00001	10269	0.0000	F900	0.00001	48416
0.0000	BA00	0.00001	10865	0.0000	FA00	0.00001	49012
0.0000	BB00	0.00001	11461	0.0000	FB00	0.00001	49608
0.0000	BC00	0.00001	12057	0.0000	FC00	0.00001	50204
0.0000	BD00	0.00001	12653	0.0000	FD00	0.00001	50800
0.0000	BE00	0.00001	13249	0.0000	FE00	0.00001	51396
0.0000	BF00	0.00001	13845	0.0000	FF00	0.00001	51992

Table F-4. (Continued)

HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL
0.0000 0000	0.00000 00000	0.0000 0040	0.00000 00149
0.0000 0001	0.00000 00002	0.0000 0041	0.00000 00151
0.0000 0002	0.00000 00005	0.0000 0042	0.00000 00154
0.0000 0003	0.00000 00007	0.0000 0043	0.00000 00156
0.0000 0004	0.00000 00009	0.0000 0044	0.00000 00158
0.0000 0005	0.00000 00012	0.0000 0045	0.00000 00161
0.0000 0006	0.00000 00014	0.0000 0046	0.00000 00163
0.0000 0007	0.00000 00016	0.0000 0047	0.00000 00165
0.0000 0008	0.00000 00019	0.0000 0048	0.00000 00168
0.0000 0009	0.00000 00021	0.0000 0049	0.00000 00170
0.0000 000A	0.00000 00023	0.0000 004A	0.00000 00172
0.0000 000B	0.00000 00026	0.0000 004B	0.00000 00175
0.0000 000C	0.00000 00028	0.0000 004C	0.00000 00177
0.0000 000D	0.00000 00030	0.0000 004D	0.00000 00179
0.0000 000E	0.00000 00033	0.0000 004E	0.00000 00182
0.0000 000F	0.00000 00035	0.0000 004F	0.00000 00184
0.0000 0010	0.00000 00037	0.0000 0050	0.00000 00186
0.0000 0011	0.00000 00040	0.0000 0051	0.00000 00189
0.0000 0012	0.00000 00042	0.0000 0052	0.00000 00191
0.0000 0013	0.00000 00044	0.0000 0053	0.00000 00193
0.0000 0014	0.00000 00047	0.0000 0054	0.00000 00196
0.0000 0015	0.00000 00049	0.0000 0055	0.00000 00198
0.0000 0016	0.00000 00051	0.0000 0056	0.00000 00200
0.0000 0017	0.00000 00054	0.0000 0057	0.00000 00203
0.0000 0018	0.00000 00056	0.0000 0058	0.00000 00205
0.0000 0019	0.00000 00058	0.0000 0059	0.00000 00207
0.0000 001A	0.00000 00061	0.0000 005A	0.00000 00210
0.0000 001B	0.00000 00063	0.0000 005B	0.00000 00212
0.0000 001C	0.00000 00065	0.0000 005C	0.00000 00214
0.0000 001D	0.00000 00068	0.0000 005D	0.00000 00217
0.0000 001E	0.00000 00070	0.0000 005E	0.00000 00219
0.0000 001F	0.00000 00072	0.0000 005F	0.00000 00221
0.0000 0020	0.00000 00075	0.0000 0060	0.00000 00224
0.0000 0021	0.00000 00077	0.0000 0061	0.00000 00226
0.0000 0022	0.00000 00079	0.0000 0062	0.00000 00228
0.0000 0023	0.00000 00081	0.0000 0063	0.00000 00231
0.0000 0024	0.00000 00084	0.0000 0064	0.00000 00233
0.0000 0025	0.00000 00086	0.0000 0065	0.00000 00235
0.0000 0026	0.00000 00088	0.0000 0066	0.00000 00237
0.0000 0027	0.00000 00091	0.0000 0067	0.00000 00240
0.0000 0028	0.00000 00093	0.0000 0068	0.00000 00242
0.0000 0029	0.00000 00095	0.0000 0069	0.00000 00244
0.0000 002A	0.00000 00098	0.0000 006A	0.00000 00247
0.0000 002B	0.00000 00100	0.0000 006B	0.00000 00249
0.0000 002C	0.00000 00102	0.0000 006C	0.00000 00251
0.0000 002D	0.00000 00105	0.0000 006D	0.00000 00254
0.0000 002E	0.00000 00107	0.0000 006E	0.00000 00256
0.0000 002F	0.00000 00109	0.0000 006F	0.00000 00258
0.0000 0030	0.00000 00112	0.0000 0070	0.00000 00261
0.0000 0031	0.00000 00114	0.0000 0071	0.00000 00263
0.0000 0032	0.00000 00116	0.0000 0072	0.00000 00265
0.0000 0033	0.00000 00119	0.0000 0073	0.00000 00268
0.0000 0034	0.00000 00121	0.0000 0074	0.00000 00270
0.0000 0035	0.00000 00123	0.0000 0075	0.00000 00272
0.0000 0036	0.00000 00126	0.0000 0076	0.00000 00275
0.0000 0037	0.00000 00128	0.0000 0077	0.00000 00277
0.0000 0038	0.00000 00130	0.0000 0078	0.00000 00279
0.0000 0039	0.00000 00133	0.0000 0079	0.00000 00282
0.0000 003A	0.00000 00135	0.0000 007A	0.00000 00284
0.0000 003B	0.00000 00137	0.0000 007B	0.00000 00286
0.0000 003C	0.00000 00140	0.0000 007C	0.00000 00289
0.0000 003D	0.00000 00142	0.0000 007D	0.00000 00291
0.0000 003E	0.00000 00144	0.0000 007E	0.00000 00293
0.0000 003F	0.00000 00147	0.0000 007F	0.00000 00296

Table F-4. (Continued)

HEXADECIMAL		DECIMAL		HEXADECIMAL		DECIMAL	
0.0000	0080	0.00000	00298	0.0000	00C0	0.00000	00447
0.0000	0081	0.00000	00300	0.0000	00C1	0.00000	00449
0.0000	0082	0.00000	00303	0.0000	00C2	0.00000	00452
0.0000	0083	0.00000	00305	0.0000	00C3	0.00000	00454
0.0000	0084	0.00000	00307	0.0000	00C4	0.00000	00456
0.0000	0085	0.00000	00310	0.0000	00C5	0.00000	00459
0.0000	0086	0.00000	00312	0.0000	00C6	0.00000	00461
0.0000	0087	0.00000	00314	0.0000	00C7	0.00000	00463
0.0000	0088	0.00000	00317	0.0000	00C8	0.00000	00466
0.0000	0089	0.00000	00319	0.0000	00C9	0.00000	00468
0.0000	008A	0.00000	00321	0.0000	00CA	0.00000	00470
0.0000	008B	0.00000	00324	0.0000	00CB	0.00000	00473
0.0000	008C	0.00000	00326	0.0000	00CC	0.00000	00475
0.0000	008D	0.00000	00328	0.0000	00CD	0.00000	00477
0.0000	008E	0.00000	00331	0.0000	00CE	0.00000	00480
0.0000	008F	0.00000	00333	0.0000	00CF	0.00000	00482
0.0000	0090	0.00000	00335	0.0000	00D0	0.00000	00484
0.0000	0091	0.00000	00338	0.0000	00D1	0.00000	00487
0.0000	0092	0.00000	00340	0.0000	00D2	0.00000	00489
0.0000	0093	0.00000	00342	0.0000	00D3	0.00000	00491
0.0000	0094	0.00000	00345	0.0000	00D4	0.00000	00494
0.0000	0095	0.00000	00347	0.0000	00D5	0.00000	00496
0.0000	0096	0.00000	00349	0.0000	00D6	0.00000	00498
0.0000	0097	0.00000	00352	0.0000	00D7	0.00000	00501
0.0000	0098	0.00000	00354	0.0000	00D8	0.00000	00503
0.0000	0099	0.00000	00356	0.0000	00D9	0.00000	00505
0.0000	009A	0.00000	00359	0.0000	00DA	0.00000	00508
0.0000	009B	0.00000	00361	0.0000	00DB	0.00000	00510
0.0000	009C	0.00000	00363	0.0000	00DC	0.00000	00512
0.0000	009D	0.00000	00366	0.0000	00DD	0.00000	00515
0.0000	009E	0.00000	00368	0.0000	00DE	0.00000	00517
0.0000	009F	0.00000	00370	0.0000	00DF	0.00000	00519
0.0000	00A0	0.00000	00373	0.0000	00E0	0.00000	00522
0.0000	00A1	0.00000	00375	0.0000	00E1	0.00000	00524
0.0000	00A2	0.00000	00377	0.0000	00E2	0.00000	00526
0.0000	00A3	0.00000	00380	0.0000	00E3	0.00000	00529
0.0000	00A4	0.00000	00382	0.0000	00E4	0.00000	00531
0.0000	00A5	0.00000	00384	0.0000	00E5	0.00000	00533
0.0000	00A6	0.00000	00386	0.0000	00E6	0.00000	00536
0.0000	00A7	0.00000	00389	0.0000	00E7	0.00000	00538
0.0000	00A8	0.00000	00391	0.0000	00E8	0.00000	00540
0.0000	00A9	0.00000	00393	0.0000	00E9	0.00000	00542
0.0000	00AA	0.00000	00396	0.0000	00EA	0.00000	00545
0.0000	00AB	0.00000	00398	0.0000	00EB	0.00000	00547
0.0000	00AC	0.00000	00400	0.0000	00EC	0.00000	00549
0.0000	00AD	0.00000	00403	0.0000	00ED	0.00000	00552
0.0000	00AE	0.00000	00405	0.0000	00EE	0.00000	00554
0.0000	00AF	0.00000	00407	0.0000	00EF	0.00000	00556
0.0000	00B0	0.00000	00410	0.0000	00F0	0.00000	00559
0.0000	00B1	0.00000	00412	0.0000	00F1	0.00000	00561
0.0000	00B2	0.00000	00414	0.0000	00F2	0.00000	00563
0.0000	00B3	0.00000	00417	0.0000	00F3	0.00000	00566
0.0000	00B4	0.00000	00419	0.0000	00F4	0.00000	00568
0.0000	00B5	0.00000	00421	0.0000	00F5	0.00000	00570
0.0000	00B6	0.00000	00424	0.0000	00F6	0.00000	00573
0.0000	00B7	0.00000	00426	0.0000	00F7	0.00000	00575
0.0000	00B8	0.00000	00428	0.0000	00F8	0.00000	00577
0.0000	00B9	0.00000	00431	0.0000	00F9	0.00000	00580
0.0000	00BA	0.00000	00433	0.0000	00FA	0.00000	00582
0.0000	00BB	0.00000	00435	0.0000	00FB	0.00000	00584
0.0000	00BC	0.00000	00438	0.0000	00FC	0.00000	00587
0.0000	00BD	0.00000	00440	0.0000	00FD	0.00000	00589
0.0000	00BE	0.00000	00442	0.0000	00FE	0.00000	00591
0.0000	00BF	0.00000	00445	0.0000	00FF	0.00000	00594

Index

- A-field, 499
- Abacus, 3
- Abort, 28, 100, 259
- Absolute value, 49, 439, 466
- ACCEPT statement, 468
- Access time, 5
- Accumulator, 19, 222, 309
- Addition, 49, 62, 309, 315, 438, 465, 592, 595
- Address, 18, 208, 427, 432
- Aiken, Howard, 4
- Algol, 12
- Algorithm, 8, 38, 301, 345
- Alphabet, 11
- Alphanumeric, 144
- Alphanumeric CRT, 249
- Analog, 4
- Analytical engine, 4
- APL, 12
- Arctangent, 49, 439, 467
- Argument, 50, 176, 188, 456, 505, 507, 584
- Arithmetic assignment statement, 48, 50, 59, 307, 440, 462
- Arithmetic comparison, 535
- Arithmetic data, 527
- Arithmetic expressions, 49, 60, 439, 465, 485
- Arithmetic IF, 485
- Arithmetic operations, 49, 146, 535
- Arithmetic operators, 308, 438, 465
- Arithmetic progression, 67
- Arithmetic unit, 18
- Array, 115, 180, 336, 429, 450, 494, 499, 579
- Array expressions, 535, 583
- Ars Magna*, 2, 4
- ASCII, 216
- Assembly language, 231, 254, 308, 413
- Assigned GOTO, 520
- Assignment operator, 50
- Associative property, 355
- AT END option, 82
- Attributes, 526
- Automatic allocation, 570
- Auxiliary storage, 5, 26
- Babbage, Charles, 4
- Back substitution, 388
- Backup, 327
- Bacon, Francis, 3
- Base, 204
- Base attribute, 527
- Basic, 12, 13, 31, 50, 120, 137, 437
- Batch-processing system, 26, 100, 101, 257, 437, 480
- BCDIC, 216

- Binary, 204
- Binary arithmetic, 592
- Binary data, 530, 531
- Binary number system, 587
- Binary point, 206
- Binomial coefficients, 111
- Bisection, 363
- Bit, 208
- Bit comparison, 535
- Bit-string data, 533
- Bit-string format, 549
- Bit-string operations, 536
- BLOCK DATA, 520
- Branch, 264
- Branch instructions, 415, 423
- Bug, 23
- Business systems, 322
- Byte, 208, 427, 432

- Call by address, 188, 507
- Call by value, 188, 507
- CALL EXIT statement, 476
- CALL statement, 512
- Capital-recovery factor, 110
- Card punch, 243
- Card reader, 18, 243
- Carriage control, 476, 498, 539, 549
- Cathode ray tube unit, 34, 249
- Central processing unit, 17, 20, 26, 33, 234
- Channel, 237
- Character, 216
- Character comparison, 535
- Character data, 143, 453, 499
- Character set, 27, 523
- Character-string data, 532
- Character-string format, 548
- Characteristic, 213
- Chord, 68
- Circular segment, 68
- Clear, 148
- Closed function, 175
- Cobol, 12, 50, 219, 266, 330
- Collective names, 574
- Comment card, 477
- Comments, 417, 444
- Common difference, 67
- Common ratio, 68
- COMMON statement, 493, 516
- Comparison operations, 535
- Compilation error, 25
- Compilation phase, 21, 46
- Compiler, 46, 254
- Complement, 210, 594
- Compound conditional, 483
- Compound conditions, 63
- Computer, 2, 20
- Computed GOTO, 485
- Concentration, 146, 536
- Condition codes, 228, 415, 423
- Conditionals, 52, 63
- Constant, 46, 437, 463, 525
- Contextually known names, 573
- Continuation card, 477
- CONTINUE statement, 490
- Control card, 28, 258, 480
- Control statements, 558
- Controlled storage, 570
- Conversion, 587
- Conversion tables, 598
- Cosine, 49, 439, 467
- Criterion function, 368
- Cryptogram, 148

- Data, 18
- Data attributes, 526
- Data base, 323
- Data card, 15, 480
- Data-directed mode, 538, 539
- Data set, 537
- DATA statement, 498, 501
- Debugging, 23, 96
- Decimal, 204
- Decimal arithmetic, 239
- Decimal data, 529, 531
- Decimal format, 463

- Decimal number system, 587
- Decimal numbers, 217
- Decimal point, 203
- Deciphering, 150
- Decision block, 52, 367
- Decision tree, 280
- DEF statement, 456
- Dependent variable, 396
- Depreciation, 91
- Deque, 153
- Desk calculator, 1
- Difference engine, 4
- DIM statement, 452
- DIMENSION statement, 493
- Direct access, 326, 520
- Direct memory access channel, 236
- Direct substitution, 406
- Discounted cash flow, 366
- Disk, 5, 248
- Display unit, 249
- Distributive property, 356
- Division, 49, 62, 309, 418, 425, 438, 465
- DO statement, 487, 559
- Double-declining balance method, 110
- Double-subscripted array, 140, 269
- Doubleword, 209, 418
- Drisk, 248
- Drum, 5, 247
- Dudeney, Henry E., 288
- Dynamic storage allocation, 120

- E-field, 473
- EBCDIC, 27, 216
- Eckert, J. Presper, 5
- Edit-directed mode, 538, 543
- EDSAC, 5
- Efficiency, 45
- Enciphering, 150, 455
- End-of-data, 486
- End-of-file, 334
- END = option, 486
- END statement, 444, 476, 503

- ENIAC, 5
- Entry, 178
- Equilibrate, 390
- EQUIVALENCE statement, 520
- ERR = option, 486
- Errors, 341
- Exchange sort, 126
- Executable statement, 476
- Execution error, 25
- Execution phase, 21
- Executive, 28, 255, 258
- Explicitly known names, 573
- Exponential format, 463
- Exponential function, 49, 439, 467
- Exponential notation, 437
- Exponentiation, 49, 62, 77, 309, 438, 465
- Expression, 49, 60, 121, 294, 439, 465, 526
- External sort, 323, 345

- F-field, 472
- Family tree, 264
- Fetch phase, 222
- Fibonacci sequence, 89, 136, 228, 374, 416, 424, 449, 489
- Field, 323, 470
- File, 35, 323, 329, 345, 459
- Finite arithmetic, 354
- Fixed-point format, 545
- Fixed-point hardware, 239
- Fixed-point representation, 210, 463, 526
- Figura universalis*, 3
- Floating-point format, 547
- Floating-point hardware, 239
- Floating-point representation, 212, 355, 463, 526
- Flowchart, 8, 38, 330, 340
- FOR statement, 449
- Format, 470, 486, 495, 499, 507, 545
- Format-free input/output, 467
- FORMAT statement, 470

INDEX

- 618**
- Formatted input/output, 469
 - Fortran, 11, 15, 50, 191, 462
 - Function, 49, 174, 176, 320, 439, 456, 466, 503
 - FUNCTION statement, 507
 - Game, 264, 275
 - Gauss reduction, 382
 - Geometric progression, 68
 - Gerbert, 3
 - GET statement, 538
 - Global, 188
 - Golden section, 368
 - GOSUB statement, 457
 - GOTO statement, 446, 483
 - Grundy's game, 287
 - Gunter, Edmund, 3
 - Guide, 523
 - H-field, 474
 - Halfword, 209
 - Hardware, 234
 - Hexadecimal, 204
 - Hexadecimal number system, 589, 594
 - Hierarchy, 60, 66, 438, 465
 - Hollerith field, 474
 - Hyperbolic functions, 69
 - I-field, 470
 - IBM System/360, 414
 - Identifiers, 524
 - IF statement, 16, 483, 485, 503, 558
 - IF-THEN statement, 14, 446
 - Implicit declaration, 560
 - Implicitly known names, 573
 - Implied DO loop, 495
 - Independent variable, 396
 - Index, 84, 449, 488
 - Index register, 229
 - Index operation, 148
 - Indexed addressing, 229, 427, 432
 - Indexing parameters, 449, 488
 - Infix notation, 296
 - Infix operator, 535
 - INPUT statement, 14, 440, 445
 - Input/output, 59
 - Input/output bus, 235
 - Input/output operations, 414, 422
 - Input/output processor, 234, 18, 26, 33
 - Input/output statements, 132
 - Instruction, 18, 222, 254, 309, 413
 - Instruction address register, 222
 - INT function, 56, 439, 465
 - Integer, 210, 463
 - INTEGER statement, 493, 508
 - Integration, 375
 - Internal procedure, 176
 - Internal sort, 323
 - Interpreter, 20, 46
 - Interrupt, 240
 - Interval halving, 363
 - Intrinsic function, 439
 - Invoked procedure, 566
 - Invoking procedure, 552, 566
 - Item, 323
 - Iteration counter, 79, 83, 449, 487
 - Job, 28
 - Keypunch, 28, 327, 341
 - Kilobyte, 210
 - Kiloword, 210
 - Known names, 573
 - Label, 415, 423, 428, 433
 - Label data, 534
 - Labeled common, 520
 - Leibniz, 4
 - LET statement, 440, 454
 - Light pen, 254
 - Line number, 13, 444
 - Line printer, 18, 26, 243, 476
 - Linear algebraic equations, 383
 - Linear list, 150
 - Linear regression, 395
 - Linkage, 175
 - Linked list, 153

- List, 268
- List-directed mode, 538, 541
- Load-and-go compiler, 21, 257
- Load phase, 22
- Loader, 22, 256, 259
- Local, 188
- Logarithm, 3, 49, 439, 467
- Logic machine, 2
- Logical file, 323
- Logical IF, 16, 483, 503
- Logical operator, 63, 484
- Loop, 39, 79, 115, 488
- Loyd, Sam, 287
- Lull, Ramon, 2, 4
- Machine language, 254
- Maclaurin series, 359
- Macro, 413
- Magnetic cores, 5, 208
- Magnetic tape, 22, 243, 257, 322, 327, 340
- Main procedure, 552
- Mark I, 4
- MAT statement, 459
- Matrix algebra, 460
- Mauchly, John, 5
- Maximum, 368
- Memory, 18, 26, 33, 234, 341, 427, 432
- Memory allocation, 119
- Merge, 332
- Mixed-mode arithmetic, 465
- Mnemonic, 19
- Modular approach, 174
- Monitor, 28, 258
- Multimodal function, 369
- Multiple branching, 102, 447
- Multiple entry, 194, 520
- Multiple return, 195, 520
- Multiple subscripts, 139
- Multiplexer channel, 237
- Multiplication, 49, 62, 309, 315, 418, 425, 438, 465
- Multiprogramming, 260
- Name, 48, 438, 464
- Namelist, 520
- Napier, John, 3
- Natural tree search, 274
- Negation, 63, 438, 465
- Nested loops, 93
- Net present value, 365
- Network, 264, 283
- NEXT statement, 449
- Node, 150, 264, 323
- Nonexecutable statement, 476
- Nonlinear algebraic equations, 362
- Nonsequential order, 325
- Normalized, 213
- Number system, 203
- Numbering rods, 3
- Numerical integration, 375
- Object deck, 22, 255
- Observation, 395
- OCCURS clause, 336
- Octal, 204
- Octal number system, 589, 594
- Offspring, 266
- ON-GOTO statement, 447
- On-line, 326
- One-way list, 153, 270
- Ones complement, 594
- Open function, 125
- Operating system, 28
- Operation code, 222
- Operator's console, 234
- Optimization, 368
- Overflow, 212, 215
- Overpunch, 219
- Packed format, 219
- Paper tape, 243
- Parabola, 69, 379
- Parallel, 208
- Parameters, 584
- Parent node, 266
- Parentheses, 62, 65, 303
- Parity, 237

- Partition, 260
- Party-line approach, 236
- Pascal, Blaise, 4
- Pedigree, 265
- Peripherals, 241
- Physical file, 323
- PL/I, 12, 50, 266, 330, 523
- Plotter, 249
- Point of invocation, 566
- Pointer, 325
- Polish string, 296
- Polygon, 69
- Port, 235, 237
- Postfix notation, 296
- Precision attribute, 527
- Prefix operator, 535
- Prime number, 41
- PRINT statement, 16, 442, 469, 470
- Problem-oriented language, 254
- Procedure, 174, 564
- Program, 7, 10
- Program counter, 222
- Program interrupt, 240
- Programmer, 10
- Programming language, 11, 20
- Punched cards, 1, 22, 27, 340
- PUT statement, 538

- Queue, 260, 153

- Radix, 204
- Random access, 326
- Range of DO, 489
- Read-only memory, 239
- READ operation (on memory), 19
- READ statement, 15, 441, 445, 467, 470, 486
- Real, 463
- REAL statement, 493, 508
- Real-time, 240, 327
- Recognition of names, 573
- Record, 266, 323
- Record declaration, 330
- Record-oriented data, 537

- Recursive, 264
- Refresh-mode CRT, 251
- Register, 230, 413
- Regression, 395
- Relational expression, 52, 63, 446, 454, 483
- Relational operators, 52, 146, 447, 483
- REM statement, 444
- Replacement operator, 50, 307, 440, 465
- Replacement sort, 123
- Return, 178
- RETURN statement, 457, 507, 512
- Right-justification, 470
- Root, 264, 363
- Rounding, 54
- Row interchange, 388

- Scalar expressions, 535
- Scale attribute, 527
- Scientific notation, 437
- Scope of names, 573
- Search, 274
- Sector, 248
- Selector channel, 237
- Sequence check, 334
- Sequential, 7, 325
- Series, 358
- Service routine, 240
- SGN function, 58, 439
- SHARE, 523
- Simple variable, 115
- Simpson's rule, 379
- Simultaneous equations, 383
- Sine, 49, 439, 467
- Single-subscripted array, 117, 336
- Slash, 497
- Slice, 31, 262
- Slide rule, 4
- Software, 234
- Sort, 123, 323, 336
- Source program, 22, 255
- Specification statements, 493

- Square root, 49, 439, 466
- Stack, 153, 275, 301, 309
- Standard deviation, 160
- Statement, 13
- Statement format, 417, 477
- Statement function, 505
- Statement-label constant, 534
- Statement-label data, 517
- Statement-label variable, 534
- Statement number, 15, 470, 477
- Stibitz, George, 5
- STOP statement, 16, 22, 447, 476, 508
- Storage allocation, 570
- Storage-tube CRT, 251
- Storage locations, 46, 208
- Store operation (on memory), 19
- Stream-oriented data, 537
- String, 148
- String data, 527, 532
- Structure expressions, 535
- Structures, 575
- Subitem, 324
- Subprogram, 176, 256, 503
- Subroutine, 183, 457, 512, 585
- SUBROUTINE statement, 512
- Subscript, 121, 180, 450
- Subscripted variables, 115, 450, 493
- Subtraction, 49, 62, 309, 438, 465, 593, 596
- Subtree, 204
- Sum of squares, 397
- Sum of year's digits method, 91
- Syntax, 10
- System chart, 322, 340
- System design, 345
- Table look-up, 128
- Tally, 341
- Teletype, 18, 33, 241
- Tens complement, 594
- Terminal, 33, 101
- Terminal node, 266
- Tic-Tac-Toe, 276
- Time-sharing, 31, 100, 101, 262, 437, 481
- Time slice, 31, 262
- Trace, 14, 101, 244
- Trailer, 80
- Transfer of Control, 446, 482
- Translation, 21
- Transportation problem, 282
- Trapezoid rule, 378
- Tree, 264, 294
- Tree search, 274
- Truncation, 54
- Turnaround, 26
- Twos complement, 594
- Typewriter, 241
- Unary minus sign, 63, 320
- Underflow, 215, 356
- Unimodal function, 369
- Vacuous loop, 86
- Variable, 46, 437, 464, 525
- Verifier, 327
- von Neumann, John, 5
- WATFIV, 480
- Word, 18, 208, 427, 432
- WRITE statement, 470
- Wynn-Williams, C.E., 5
- Wythoff, W.A., 286
- XDS Sigma 5/7, 422
- Zero subscript, 452
- Zoned format, 217



0700224203

09/12/2017 11:37-3

22

016600